基于TensorFlow深度学习，掌握自然语言处理及其应用

- 自然语言处理与深度学习基础、Skip-Gram、CBOW、GloVe
- 基于卷积神经网络的句子分类、循环神经网络、LSTM
- 基于LSTM生成文本的应用，基于LSTM进行中文情感分析
- 机器翻译、图像字幕自动生成、智能问答系统





TensorFlow与自然语言处理应用

李孟全 著

清华大学出版社



TensorFlow与自然语言处理应用

李孟全 著

清华大学出版社
北 京

内 容 简 介

自然语言处理（NLP）是计算机科学、人工智能、语言学关注计算机和人类（自然）语言之间的相互作用的领域。自然语言处理是机器学习的应用之一，用于分析、理解和生成自然语言，它与人机交互有关，最终实现人与计算机之间更好的交流。

本书分为 12 章，内容包括自然语言处理基础、深度学习基础、TensorFlow、词嵌入（Word Embedding）、卷积神经网络（CNN）与句子分类、循环神经网络（RNN）、长短期记忆（LSTM）、利用 LSTM 实现图像字幕自动生成、情感分析、机器翻译及智能问答系统。

本书适合 TensorFlow 自然语言处理技术的初学者、NLP 应用开发人员、NLP 研究人员，也适合高等院校和培训学校相关专业的师生教学参考。

图书在版编目（CIP）数据

TensorFlow 与自然语言处理应用 / 李孟全著.—北京：清华大学出版社，2019

ISBN 978-7-302-53101-2

Ⅰ. ①T…　Ⅱ. ①李…　Ⅲ. ①人工智能—算法—应用—自然语言处理　Ⅳ. ①TP391

中国版本图书馆 CIP 数据核字（2019）第 101114 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：三河市龙大印装有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**26　　**字　　数：**665 千字
版　　次：2019 年 8 月第 1 版　　**印　　次：**2019 年 8 月第 1 次印刷
定　　价：89.00 元

产品编号：082163-01

前　言

2018年，其实是自然语言处理领域收获颇丰的一年，尤其是以Google公司在2018年11月份发布的BERT模型最为世人所瞩目，可以说是最近AI研究领域最为火爆的历史性突破。最近几年，无论从媒体报道还是切身感受，我们都看到人工智能目前的发展势头非常迅猛。如果我们简单回顾一下人工智能发展历程，不难发现其轨迹有三个发展阶段：第一个阶段是计算智能阶段，其典型表现在于计算机和人类相比是能存会算，其超大存储量、超高计算速度方面均可完胜我们人类；第二个阶段是感知智能阶段，具体表现在以语音识别和图像识别技术为代表的迅猛发展，如综艺电视节目中的"机智过人""最强大脑"就是其很好的呈现形式；第三个阶段是认知智能阶段，这个阶段需要机器能够思考并具有情感。正因为人工智能与我们日常生活的联系越来越紧密，且自然语言处理技术是推动机器实现认知的关键性研究领域，所以我们有必要对自然语言处理应用进行深入探索。本书将利用目前流行的Google技术框架（TensorFlow）来实现自然语言处理方面的应用。

由于我们生活的方方面面被赋予了越来越多的数字化内容，因此相应的数据量也在呈指数级增长，并且大多数数据是与语言相关的数据，如电子邮件、社交媒体帖子、电话和网络文章，自然语言处理（NaturalLanguageProcessing，NLP）能够有效地利用这些数据帮助人们完成日常业务工作。NLP 已经彻底改变了我们使用数据改善业务和生活的方式，并将在我们未来的日常生活中发挥更大的作用。

NLP最普遍的使用案例之一是虚拟助手（Virtual Assistants，VA），如百度小度助手、Apple的Siri、谷歌助手（Google Assistant）和亚马逊 Alexa。当我向VA询问"附近最便宜的火锅"时（笔者利用手机上百度地图小度时，它会把附近最便宜的火锅店排在第一位），就会触发一系列复杂的NLP任务。首先VA需要了解（解析）我的请求（了解它需要检索火锅的价格，而不是停车位计时的价格），VA做出的决定是"什么是便宜的？"；然后VA需要对附近火锅的价格进行排名（也有可能基于我过去吃过的火锅店）；最后VA抓取相关数据获取附近火锅的价格，并通过分析每家火锅店的价格和评论对它们进行排名。其实，我们在几秒钟内看到的结果是执行一系列非常复杂NLP任务的结果。

正是NLP在我们日常生活中呈现出越来越多的便利性，笔者才更想对NLP背后的模型原理和具体应用进行深入的探讨，以便我们对NLP有更多的认知。另外，笔者查看了近些年来的相关文献，发现单独讲解NLP方面的理论文献国内外都有，单独撰写NLP任务实现的技术工具（如TensorFlow）的图书也很多，而将二者结合起来的图书，目前在国内还没有发现（也许有，只是笔者没有发现而已）。于是，2018年4月，笔者就想对关于利用TensorFlow技术框架来实现NLP任

务应用方面进行成体系地探索，以便对今后的工作有所积累。本书在创作过程中参考了《*Natural Language Processing with TensorFlow*》（Thushan Ganegedara 著）中的一些内容，在此向 Thushan Ganegedara 表达个人的敬意！

通过阅读本书，你将学会如何利用深度学习来实现许多有意义的 NLP 任务。对于本书中涉及的 NLP 任务，我们都有具体的代码实现（含实现过程），使用的技术框架为 TensorFlow（1.8 版本），编程语言为 Python（3.6 版本）。

本书主要内容

第 1 章　自然语言处理基础。首先介绍自然语言处理的含义及 NLP 中的一些常见子任务；然后讲述 NLP 的发展历程：偏理论的理性主义、偏实践应用的经验主义和深度学习阶段；接着对 NLP 任务中深度学习的局限性进行了大致分析；最后，我们对于 NLP 的应用场景和发展前景做了简要阐述。

第 2 章　深度学习基础。首先介绍深度学习的概念和演变过程，同时介绍了深度学习的基础模型——神经元模型，并对单层神经网络和多层神经网络模型（深度学习）的结构和原理进行了深度解读；然后介绍 Encoder-Decoder 网络和深度学习中最常见的优化算法——随机梯度下降；最后介绍反向传播算法（BP 算法）。

第 3 章　TensorFlow。首先介绍 TensorFlow 的产生、主要特征和相关安装等基础准备内容，同时介绍了在执行 TensorFlow 程序时需要构建计算图及其构成元素，并详细解读了 TensorFlow 的架构和工作原理，还给出了一个示例来加深理解；然后从 TensorFlow 客户端的角度通过代码逐层剖析 TensorFlow 的内部运行情况，以便让我们对 TensorFlow 的内部运行机制、各个基础组件之间的关联等有深入的了解；最后介绍变量作用域机制并实现一个完整的神经网络示例。

第 4 章　词嵌入。首先介绍分布式表示，对其含义、类型及简要特征有一个直观的认识；其次，我们重点对 CBOW 模型、Skip-gram 模型及 GloVe 模型的工作原理和内部架构进行深度解析；最后利用前面的模型，通过 TensorFlow 实现一个文档分类任务。

第 5 章　卷积神经网络（CNN）与句子分类。首先介绍 CNN 的历史演变过程，并对其 5 个层级结构（输入层、卷积运算层、激励层、池化层、全连接层）和 4 个基本运算单元（卷积运算、池化运算、全连接运算和识别运算）进行了介绍；然后介绍几种常见的经典卷积神经网络：AlexNet、VGGNet、Google Inception Net 和 ResNet，并逐一从模型思想、结构、特性亮点等方面进行详细解读；最后为了将上述解析的模型思想真正落实到代码层面上，我们给出两个应用案例：利用 CNN 对 MNIST 数据集进行图片分类和利用 CNN 对句子进行分类。

第 6 章　循环神经网络（RNN）。首先通过计算图及其展开解读了循环的任何函数本质上可以认为是一种循环神经网络的说法；然后介绍时间的反向传播算法（BPTT），我们将学习反向传播的工作原理、为什么不能对 RNN 使用标准反向传播、如何使用 BPTT 对数据进行 RNN 训练、截断

BPTT 和 BPPTT 的局限性等，并解读其局限性中的常见问题及对应的解决方法；最后我们将看到一个 RNN 的应用——文本生成，并研究了一种能够捕获更长记忆的 RNN 变体(被称为 RNN-CF)，还给出了其在实例中的应用。

第 7 章　长短期记忆网络（LSTM）。首先介绍 LSTM 及其高级架构，并深入研究 LSTM 中发生的详细计算，结合一个例子讨论了该计算的过程；然后介绍了几个可以提高 LSTM 性能的扩展，即贪心采样（一种非常简单的技术）、集束搜索（一种更复杂的搜索技术）及 BiLSTM 模型等；最后介绍了标准 LSTM 的两种变体：窥视孔连接和 GRU。

第 8 章　利用 LSTM 自动生成文本。首先广泛地评估了 LSTM 在文本生成任务中的表现；然后定性和定量地测量 LSTMS 生成的文本有多好，并对 LSTM、带窥视孔连接的 LSTM 和 GRU 进行比较；最后如何将词嵌入引入模型中，以改进 LSTM 生成的文本。

第 9 章　利用 LSTM 实现图像字幕自动生成。首先回顾了图像字幕的主要发展及其对研究和行业部署的影响；其次详细介绍了基于深度学习的图像字幕自动生成的两大框架；最后对图像字幕的自动生成任务进行了详解。

第 10 章　情感分析。首先介绍情感分析的应用、情感问题的界定及情感文档的分类，同时对句子观点的主观性、基于 Aspect 的情感分析、情感词典的生成、比较观点分析及观点的检索做了介绍；然后重点阐述了垃圾评论的各种情况；最后利用 TensorFlow 对于酒店评论样本数据进行了情感分析建模比较，并得出相关结论。

第 11 章　机器翻译。首先对基于规则的机器翻译、统计机器翻译等传统的机器翻译情况进行了详细解释，并对基于神经网络的神经网络机器翻译模型的架构和工作机制进行了深度剖析，同时对 2018 年 11 月份发布的具有重大突破性的 BERT 模型进行了分析；然后介绍如何实现一个 NMT 系统；最后讲解如何改进标准 NMT 系统。

第 12 章　智能问答系统。本章简要介绍了基于深度学习的问答方法，特别是对知识库和机器理解的问答。深度学习的优点是可以将所有文本跨度（包括文档、问题和潜在答案）转换为向量嵌入，然而基于深度学习的 QA 模型存在许多挑战。例如，现有的神经网络（RNN 和 CNN）仍然不能精确地捕获给定问题的语义含义，特别是对于文档，主题或逻辑结构不能通过神经网络容易地建模，并且在知识库中嵌入项目仍然没有有效的方法，以及 QA 中的推理过程很难通过向量之间的简单数值运算来建模。这些问题是质量保证任务面临的主要挑战，未来应引起更多的关注。

代码下载与技术支持

本书示例代码下载地址请扫描下面的二维码获得。如果发现书中存在问题或对本书有什么建议，请联系电子邮箱 **280751474@qq.com**。



致谢

笔者在撰写这样一本技术性很强的书时确实有不少感慨，但正如一位技术界的前辈说的“没有等出来的美丽，只有走出来的辉煌”，所以在人生的路上，能够让自己用行动对冲遗憾，足矣。

在此，我要感谢本书的每一位读者，希望本书能够为大家带来一些启发。感谢清华大学出版社夏毓彦及其他老师的支持。感谢周一铁先生、张述睿先生的支持。感谢我的朋友和同学们，得益于你们一直以来的理解和支持，我才能有信心完成这本书的创作。

笔者自认才疏学浅，对深度学习和自然语言处理也仅是略知皮毛，且因时间有限，书中难免有错谬之处，还请各位读者予以告知，将不胜感激！

李孟全
2019 年 5 月

目　录

第 1 章 自然语言处理基础

自然语言处理（Natural Language Processing，NLP）是一个跨学科领域，它结合了计算科学、语言学、认知科学和人工智能，主要研究能够让计算机实现与人类语言有关的各类任务的各种理论和方法，特别是如何对计算机进行编程以处理和分析大量自然语言数据（非结构化的数据）。从科学的角度来看，NLP 旨在对人类语言理解和产生的认知机制进行建模。从工程角度来看，NLP 关注如何开发新颖的实际应用程序以促进计算机与人类语言之间的交互。在自然语言处理中，经常遇到的挑战包括语音识别、口语理解、对话系统、词汇分析、句法解析、机器翻译、知识图谱、信息检索、问题问答、情感分析、社交计算、自然语言生成和自然语言摘要等。当然，自然语言处理工作也是计算机科学中极其困难的工作任务。语言本身存在着各种各样的问题，亦因语言而异。

幸运的是，最近几年深度学习领域获得快速发展，使得深度学习算法在诸如图像分类、语音识别、文本生成、机器翻译等诸多带有很强挑战性的工作任务中表现优异，加速了深度学习与 NLP 各工作任务的深度融合，从而使得自然语言处理领域焕发出新的活力。而在深度学习被广泛应用的过程中，出现了多种技术框架，其中 TensorFlow 是目前最直观、最有效的深度学习框架之一。本书重点探讨如何利用 TensorFlow 深度学习框架去实现 NLP 的各种任务，例如句子分类、文档分类、文本生成、图像字幕自动生成、机器翻译、智能问答等。

在本章中，我们将要对于自然语言处理基础有一个初步了解，并对 NLP 的主要工作任务做一个划分；然后我们将对 NLP 领域的三个发展浪潮做详细解读，并对当前 NLP 领域中深度学习的局限性进行剖析；最后，我们还会对于 NLP 的应用场景和应用前景做个简要阐述。

1.1 认识自然语言处理

根据《2017 微信数据报告》显示，每天会有 380 亿条信息从微信上发出，如果按照每条信息都是文字“你吃饭了么”计算，通过微信发送的信息每天的数据量在 350GB 以上（一个汉字占 2 字节，1024 字节 1KB）。而实际的数据量会更多，因为这些信息会有不少语音、动画表情、图片、小视频等。其实，在实际工作中，我们每天都在处理的电子邮件、各类报告文档等同样也在以惊人的速度充斥着整个网络环境。2018 年 6 月，据科技公司 Domo 预测，到 2020 年，世界上每人每天将产生超过 140GB 的数据，并且随着物联网的迅猛发展，这个数字将会继续扩大。

正是由于这些统计数据的存在，才使得我们为界定 NLP 提供了良好的基础。简而言之，NLP 的目标就是让机器拥有真正理解人类语言并以与人类相同的方式处理它的能力。如今，NLP 的应用已经广泛存在，就像我们日常生活中常用到的虚拟助手（VA），常见的有百度语音助手、讯飞语音助手、Google 智能助理、微软的个人智能助理小娜（Cortana）、苹果系统的 Siri 等，这些虚拟助手主要是 NLP 系统在运行。比如，你告诉语音助手“请告诉我附近好吃的麻辣烫在哪儿？”首先，VA 需要将你的声音转换为文本（语音到文本）。接下来，VA 必须理解你请求的语义（例如，你正在寻找带有麻辣烫美食且好吃的餐厅）并制定结构化请求（例如，美食=麻辣烫，评级= 3-5，距离< 3 公里）。然后，VA 必须按位置和菜肴两个条件搜索并筛选出餐厅，再按收到的评级对餐厅进行排序。为了计算餐厅的整体评级，良好的 NLP 系统可以查看每个用户提供的评级和文本描述。最后，用户到达餐厅，VA 还可以将各种菜品组合的受欢迎程度进行综合推荐，以此来帮助你做出更好的选择。这个例子表明 NLP 已成为人类生活中不可或缺的一部分。

1.2 自然语言处理方面的任务

NLP 其实有许多实际的应用，而一个好的 NLP 系统会执行多个任务系统。比如，上面提到的你要在当前位置选择麻辣烫小吃店的例子，其实就是在执行多个 NLP 任务系统。关于 NLP 的任务，主要有以下几类：

- **标记化：** 标记化是将文本语料库分离为原子单元（例如，单词）的任务。虽然看似微不足道，但是标记化却是一项非常重要的工作任务。例如，在日语中，单词不以空格或标点符号分隔。
- **词义消歧（Word-sense Disambiguation，WSD）：** 词义消歧是识别单词正确含义的任务。例如，有两个句子，“你提供的图真好看”和“你图啥？”，其中“图”就有两种不同的含义。词义消歧对于诸如问答之类的任务至关重要。
- **命名实体识别（Named Entity Recognition，NER）：** NER 尝试从给定的文本主体或文本语料库中提取实体（例如，人、位置和组织）。例如，有一个句子，“林阿姨昨天在小区门口给了小明两瓶牛奶”，将被转换为 *林阿姨*$_{\text{人}}$ *昨天*$_{\text{时间}}$ 在 *小区门口*$_{\text{位置}}$ 给了

小明$_{\text{人}}$*两瓶*$_{\text{数量}}$牛奶。NER 是信息检索和知识表示等领域的一个重要课题。

- **词性（Part-of-Speech，PoS）标注**：是词汇基本的语法属性，通常也称为词类，既可以是名词、动词、形容词、副词、介词等基本标签，也可以是诸如专有名词、普通名词、短语动词等。词性标注就是在给定句子中判定每个词的语法范畴，确定其词性并加以标注的过程，是中文信息处理面临的重要基础性问题，主要可以分为基于规则和基于统计的方法。
- **句子/概要分类**：句子或概要（例如，电影评论）分类有许多用例，例如垃圾邮件检测、新闻文章分类（诸如政治、科技和体育等）和产品评论评级（正面或负面）。这是通过训练带标签的数据（由人类注释的评论，带有积极或消极的标签）来训练分类模型实现的。
- **文本生成**：在文本生成中，学习模型（例如，神经网络）使用文本语料库（大量文本文档集合）进行训练，预测随后的新文本。例如，文本生成可以通过使用现有的小说故事文本进行训练来输出一个全新的小说故事文本。当然，具体的实现过程会涉及具体模型的实施，具有一定的复杂性。本书第 8 章将专门针对文本生成做详细解读。
- **问答（QA）系统**：QA 技术具有很高的商业价值，因为这些技术是聊天机器人和 VA（例如谷歌 Assistant 和苹果 Siri）实现的基础所在。聊天机器人已经被许多公司用于客户支持工作。聊天机器人可以用来回答和解决客户直接关心的问题，而不需要人工干预。QA 涉及 NLP 的许多方面，比如信息检索和知识图谱中的知识表示。因此开发 QA 系统变得更加具有挑战性。
- **机器翻译**：是将一个句子/短语从源语言（如汉语）转换为目标语言（如英语）的任务。这是一个非常具有挑战性的任务，因为不同的语言具有高度不同的形态结构，这意味着它不是一对一的转换。此外，语言之间的字对字关系可以是一对多、一对一、多对一或多对多。这在 MT 文献中被称为单词对齐问题。

为了开发一个可以帮助人们完成日常任务的系统（例如，VA 或聊天机器人），这些任务中的许多工作需要放在一起执行。正如我们在前面的例子中看到的那样，“请告诉我附近好吃的麻辣烫在哪儿？”需要完成几个不同的 NLP 任务，例如语音到文本转换、语义和情感分析、问题回答和机器翻译。在图 1.1 中，我们提供了不同 NLP 任务的层次分类。我们首先有两大类任务：分析（分析现有文本）和生成（生成新文本）任务。然后将分析分为三类：句法（基于语言结构的任务）、语义（基于意义的任务）和语用（难以解决的开放问题），如图 1-1 所示。

目前，我们对于自然语言处理及其各种任务分类有了一定的了解，下面我们将探讨一下 NLP 的起源、发展及现状。

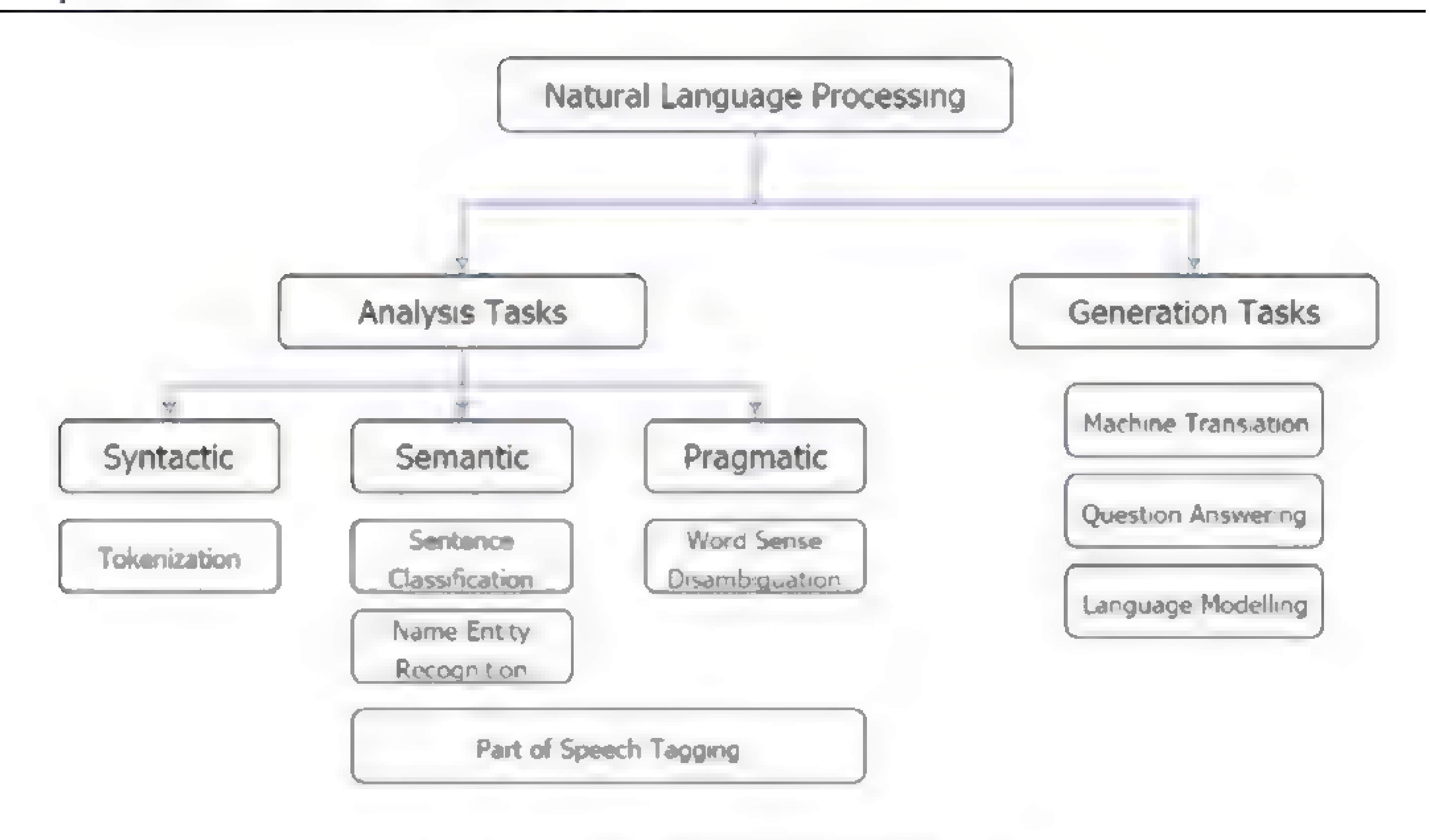


图 1-1 广义范畴下的 NLP 主要任务分类

1.3 第一阶段：偏理论的理性主义

NLP 研究的第一次浪潮持续了很长一段时间，可以追溯到 20 世纪 50 年代。1950 年，阿兰·图灵提出了图灵测试来评估计算机展示与人类无法区分的智能行为的能力（图灵，1950）（注意：本书这种“人名，年份”的说明方式用于读者必要时查阅下载资源文件，以确认参考文献的出处）。该测试基于人和计算机之间的自然语言对话，旨在产生类似人的响应。1954 年，Georgetown-IBM 实验展示了第一个能够将 60 多个俄语句子翻译成英语的机器翻译系统。

这些方法基于这样一种信念，即人类大脑中的语言知识是通过一般遗传而提前固定下来的，这在 20 世纪 60 年代到 20 世纪 80 年代后期的 NLP 研究中占主导地位。这些方法被称为理性主义方法（Church，2007）。理性主义方法在 NLP 中占有主导地位，主要是由于诺姆·乔姆斯基关于先天语言结构的论点得到广泛接受，以及他对 N-gram 的批评（Chomsky，1957）。假设语言的关键部分在出生时就已经扎根于大脑，作为人类遗传的一部分，理性主义方法会努力设计人工制作的规则，将相关知识和推理机制融入智能 NLP 系统。直到 20 世纪 80 年代，最著名的 NLP 系统是基于复杂的手写规则集的，例如模拟罗格氏（Rogerian）心理治疗师的 ELIZA 和将现实世界的信息构造成概念本体的 MARGIE，是基于复杂的手写规则集。

这一时期大致与人工智能的早期发展相吻合，人工智能以专家知识工程为特征，行业专家根据他们所拥有的（非常狭窄的）应用领域的知识设计了计算机程序（Nilsson，1982；Winston，1993）。专家们使用基于细致的表示和工程学知识的符号逻辑规则来设计这些程序。这些基于知识的人工智能系统往往通过检查“大脑”或最重要的参数，并针对每个具体情况采取适当行动，从而有效地解决特定领域的问题。这些“大脑”参数由人类专家提前确定，使“尾部”参数和案例保持不变。由

于缺乏学习能力，很难将其解决方案推广到新的场景和领域。在此期间的典型方法是专家系统，例如模拟人类专家决策能力的计算机系统。这种系统旨在通过推理知识来解决复杂问题（Nilsson，1982）。第一个专家系统创建于 20 世纪 70 年代，而后在 20 世纪 80 年代兴起。使用的主要“算法”是“if-then-else”形式的推理规则（Jackson，1998）。这些第一代人工智能系统的主要优势在于其执行逻辑推理（有限的）能力的透明性和可解释性。就像 ELIZA 和 MARGIE 这样的 NLP 系统一样，早期的专家系统使用人工制作的专家知识库，这些知识在某些特定的问题中往往是有效的，尽管推理机制不能处理实际应用中普遍存在的不确定性。

对于语音识别的研究和系统设计，NLP 和人工智能面临的一个长期挑战是在很大程度上需要依赖于专家知识工程的范式，正如 Church 和 Mercer （Church 和 Mercer，1993）所分析的那样。在 20 世纪 70 年代和 80 年代初期，语音识别的专家系统方法非常受欢迎(Reddy，1976；Zue，1985)。然而，研究人员敏锐地认识到该阶段缺乏从数据中学习和处理推理中不确定性的能力，继而出现接下来描述的第二阶段语音识别、NLP 和人工智能。

1.4　第二阶段：偏实践应用的经验主义

该阶段 NLP 的特点是通过数据语料库和（浅）机器学习、统计或其他方法来使用数据样本（Manning 和 Schtze，1999）。由于自然语言的大部分结构和理论被数据驱动的方法所忽视或抛弃，所以这期间发展起来的主要方法被称为经验的（或实用的）方法（Church and Mercer，1993；Church，2014）。随着机器可读数据可用性的增加和计算能力的不断提高，从 1990 年开始，经验方法一直主导着 NLP。其中一个主要的 NLP 会议甚至被命名为“自然语言处理中的经验方法（EMNLP）”，以最直接地反映出 NLP 研究人员在该阶段对经验方法的强烈（积极）倾向性。

与理性主义方法相反，经验方法假设人类思维只从联想、模式识别和概括的一般操作着手。为了使得大脑更好地学习自然语言的详细结构，需要存在丰富的感官输入才可以。自 1920 年以来，经验主义在人口学中普遍存在，自 1990 年以来经验主义也一直在复苏。早期的 NLP 经验方法侧重于开发生成模型，如隐马尔可夫模型（HMM）（Baum 和 Petrie，1966）、IBM 翻译模型（Brown 等，1993）和脑部驱动的解析模型（Collins，1997）从大型语料库中发现语言的规律性。自 20 世纪 90 年代末以来，判别模型已成为各种 NLP 任务中实用的方法。NLP 中的代表性判别模型和方法包括最大熵模型（Ratnaparkhi，1997）、支持向量机（Vapnik，1998）、条件随机场（Lafferty 等，2001）、最大互信息和最小分类误差（He 等，2008）和感知器（Collins，2002）。

同样，NLP 中的经验主义时代与人工智能以及语音识别和计算机视觉中的方法相对应。这是因为有明确的证据表明，学习和感知能力对于复杂的人工智能系统至关重要，但在前一波流行的专家系统中却缺失了。例如，当 DARPA 开启其首次自动驾驶大挑战时，大多数车辆依赖于基于知识的人工智能范式。与语音识别和 NLP 非常相似，自动驾驶和计算机视觉研究人员立即意识到基于知识范式的局限性，因为机器学习必须具有不确定性处理和泛化能力。

NLP 中的经验主义和第二阶段中的语音识别是基于数据密集型的机器学习，我们现在称之为“浅层”机器学习，因为这里通常会缺少由多层或“深层”数据表示构成的抽象，第三阶段深度学

习方面将在后面继续开展。在机器学习中，研究人员无须关注构建第一阶段期间基于知识的 NLP 和语音系统所需的精确度和正确规则。他们关注统计模型（Bishop，2006；Murphy，2012）或简单的神经网络（Bishop，1995）作为潜在引擎。然后，他们使用充足的训练数据自动学习或“调整”引擎的参数，以使它们处理不确定性，并尝试从一个场景推广到另一个场景，从一个域到另一个域。用于机器学习的关键算法和方法包括 EM、贝叶斯网络、支持向量机、决策树及用于神经网络的反向传播算法。

现在回过头来看，基于机器学习的 NLP、语音识别和其他人工智能系统，比早期的基于知识的对应部分表现更佳。诸如一些成功的例子，包括机器知觉中的几乎所有人工智能任务——语音识别（Jelinek，1998）、人脸识别（Viola 和 Jones，2004）、视觉对象识别（Fei-Fei 和 Perona，2005）、手写识别（Plamondon 和 Srihari，2000）和机器翻译（Och，2003）。

具体来看，针对机器翻译应用方面，传统方法还是以统计方法主，我们也会在本书的第 11 章对机器翻译部分做详细解读

双语训练数据中句子级对齐的可用性使得不通过规则而是直接从数据中获得表层翻译成为可能，代价是丢弃或忽略自然语言中的结构化信息。当然，在本阶段的后续发展中，机器翻译的质量也得到了显著提升（Och 和 Ney，2002；Och，2003；Chiang，2007；He 和 Deng，2012），但还是没有达到现实世界中大规模部署的水平（后续深度学习阶段将会继续探讨）。

在 NLP 的对话和口语理解领域，这个经验主义时代也以数据驱动的机器学习方法为显著标志，这些方法非常适合于定量评价和具体可交付成果的要求。他们关注的是文本和域的更广泛但肤浅的表层覆盖，而不是对高度受限的文本和域的详细分析。我们训练数据的目的，不是从对话系统中设计出有关语言理解和动作反映方面的规则，而是从数据样本中自动学习（浅层）统计或神经模型方面的参数。这种学习有助于降低人工制作复杂对话管理器的设计成本，并有助于提高整体口语理解和对话系统中语音识别错误的鲁棒性水平（He 和 Deng，2013）。具体来看，对话系统中对话策略部分，在本阶段引入了基于马尔科夫决策过程的强化学习，有关评论，可以参阅 Young 等人的文章（Young 等，2013）。在口语理解方面，主要方法从第一阶段基于规则或模板的方法转移到生成模型，如隐马尔科夫模型（HMMs）（Wang 等，2011），再到判别模型，如条件随机场（Tur 和 Deng，2011）。

同样，在语音识别领域，从 20 世纪 80 年代早期到 2010 年左右，该领域主要由机器学习（浅）范式主导，使用基于与高斯混合模型集成的 HMM 的统计生成模型，以及不同版本的泛化方面（Baker 等，2009；Deng 和 O'Shaughnessy，2003；Rabiner 和 Juang，1993）。广义 HMMs 的许多版本是基于统计和神经网络的隐藏动态模型（Deng，1998；Bridle 等，1998；Deng 和 Yu，2007)。前者采用 EM 和扩展卡尔曼滤波算法来学习模型参数（Ma 和 Deng，2004；Lee 等，2004）；后者使用了反向传播（Picone 等，1999）。它们都广泛地利用了多个潜在的表示层来生成语音波形，遵循人类语音感知中长期存在的通过合成进行分析的框架。更重要的是，将这种“深层”生成过程转化为端到端判别过程的对应，引起了深度学习第一次在工业上的成功（Deng 等，2010，2013；Hinton 等，2012），形成了第三阶段的语音识别和 NLP 的驱动力，接下来我们将对此进行阐述。

1.5 第三阶段：深度学习阶段

虽然在第二阶段开发的 NLP 系统，包括语音识别、语言理解和机器翻译，比第一阶段开发系统的表现更好，具有更高的鲁棒性，但它们远未达到人类级别的水平，还有很多地方需要改进。除了少数例外，NLP 的（浅）机器学习模型通常没有足够大的容量来吸收大量的训练数据。此外，涉及的这些学习算法、方法和基础结构不够强大。所有这一切都在几年前发生了很大变化，由于深层结构化机器学习或深层学习的新范式推动（Bengio，2009；Deng 和 Yu，2014；LeCun 等，2015；Goodfellow 等，2016），引发了 NLP 的第三波浪潮。

在传统的机器学习中，由于特征是由人设计的，需要大量的人类专业知识，显然特征工程也存在一些瓶颈。同时，相关的浅层模型缺乏表示能力，因此缺乏形成可分解抽象级别的能力，这些抽象级别在形成观察到的语言数据时将自动分离复杂的因素。深度学习的进步是当前 NLP 和人工智能拐点背后的主要推动力，并且直接推动了神经网络的复兴，包括商业领域的广泛应用（Parloff，2016）。

进一步讲，尽管在第二次浪潮期间开发的许多重要的 NLP 任务中，判别模型（浅层）取得了成功，但它们仍然难以通过行业专家人工设计特征来涵盖语言中的所有规则。除了不完整性问题之外，这种浅层模型还面临稀疏性问题，因为特征通常仅在训练数据中出现一次，特别是对于高度稀疏的高阶特征。因此，在深度学习出现之前，特征设计已经成为统计 NLP 的主要障碍之一。深度学习为解决我们的特征工程问题带来了希望，其观点被称为“从头开始 NLP”（Collobert 等，2011），这在深度学习早期被认为是非同寻常的。这种深度学习方法利用了包含多个隐藏层的强大神经网络来解决一般的机器学习任务，而无须特征工程。与浅层神经网络和相关的机器学习模型不同，深层神经网络能够利用多层非线性处理单元的级联来从数据中学习表示以进行特征提取。由于较高级别的特征源自于较低级别的特征，因此这些级别构成了概念上的层次结构。

深度学习起源于人工神经网络，可以将其视为受生物神经系统启发的细胞类型的级联模型。随着反向传播算法的出现（Rumelhart 等，1986），从零开始训练深度神经网络在 20 世纪 90 年代受到了广泛的关注。其实在早期，由于没有大量的训练数据，也没有适当的设计模式和学习方法，在神经网络训练期间，学习信号在层与层之间传播时会随着层数呈指数级消失，难以调整深度神经网络的连接权重值，尤其是循环模式。Hinton 等人最初克服了这个问题（Hinton 等，2006），使用无监督的预训练，首先学习通常有用的特征检测器，然后通过监督学习进一步训练网络，进而对标记数据进行分类。因此，可以使用低级表示来学习高级表示的分布。这项开创性的工作标志着神经网络的复兴。此后各种网络架构被提出并开发出来，包括深度信念网络（deep belief networks）（Hinton 等，2006）、栈式自动编码器（stacked auto-encoders）（Vincent 等，2010）、深度玻尔兹曼机（deep Boltzmann machines）（Hinton 和 Salakhutdinov，2012）、深度卷积神经网络（Krizhevsky 等，2012）、深度堆叠网络（deep stacking networks）（Deng 等，2012）以及深度 Q 网络（Mnih 等，2015）。2010 年以来，深度学习能够发现高维数据中复杂的结构，已成功应用于人工智能的各种实际任务中，尤其是语音识别（Yu 等，2010；Hinton 等，2012）、图像分类（Krizhevsky 等，

2012；He 等，2016）和 NLP。

语音识别是 NLP 的核心任务之一，且它在工业 NLP 实际应用中受到深度学习很大的影响，所以我们这里对此进行一些解读。深度学习在大规模语音识别中的工业应用在 2010 年左右开始起飞。相关工作是由学术界和产业界合作发起的，最初的成果是在 2009 年 NIPS 语音识别和相关应用的深度学习研讨会上发布的。这次研讨会的目的是语音深层生成模型的局限性，以及大计算、大数据时代需要对深层神经网络进行认真研究的可能性。当时认为，使用基于对比散度学习算法（Contrastive Divergence Learning Algorithm）的深度信念网络生成模型进行 DNNs 预处理，可以克服 20 世纪 90 年代神经网络遇到的主要困难（Dahl 等，2011；Mohamed 等，2009）。然而，在微软早期关于这项的研究中，人们发现，没有对比散度预训练，而是使用大量的训练数据，连同深层神经网络，这些深层神经网络设计成具有相应的大型、上下文相关的输出层，并且经过精心的工程设计，可以获得比当时最先进的（浅）机器学习系统显著更低的识别误差（Yu 等，2010，2011；Dahl 等，2012）。北美的其他几个主要语音识别研究小组（Hinton 等，2012 年；Deng 等，2013 年）以及随后一些海外研究小组很快就证实了这一发现。此外，还发现这两种类型系统产生的识别错误的本质是不同的，这为如何将深度学习集成到现有的由主要参与者在语音识别中部署的高效运行的语音解码系统提供了技术支持（Yu 和 Deng，2015；Abdel-Hamid 等，2014；Xiong 等，2016；Saon 等，2017）。如今，应用于各种形式的深层神经网络的反向传播算法被统一应用于所有当前最先进的语音识别系统（Yu 和 Deng，2015；Amodei 等，2016；Saon 等，2017），以及所有主要的商业语音识别系统——微软 Cortana、Xbox、Skype 翻译、亚马逊 Alexa、谷歌助理、苹果 Siri、百度小度和 iFlyTek 语音搜索等——都基于深度学习方法。

2010 年、2011 年语音识别的惊人成功预示着第三波 NLP 和人工智能的到来。随着深度学习在语音识别领域的成功，计算机视觉（Krizhevsky 等，2012）和机器翻译（Bahdanau 等，2015）也很快被类似的深度学习范式所取代。特别是，虽然早在 2001 年就开发了强大的词汇神经词嵌入技术（Bengio 等，2001，Bengio 等人在 2001 年发表在 NIPS 上的文章《*A Neural Probabilistic Language Model*》，现在多数看到的是他们在 2003 年投到 JMLR 上的同名论文），但直到十多年后，由于大数据的可用性和计算机更快的计算能力，它才被证明在实际大规模场景下具有真正的价值（Mikolov 等，2013）。此外，还有大量的其他 NLP 应用，如图像字幕（Karpathy 和 Fei-Fei，2015；Fang 等，2015；Gan 等人，2017）、视觉问答（Fei-Fei 和 Perona，2016）、语音理解（Mesnil 等，2013）、网络搜索（Huang 等，2013）和推荐系统；由于深度学习的广泛应用，也有许多非 NLP 任务，像药物发现和毒理学、客户关系管理、手势识别、医学信息学、广告、医学图像分析、机器人、无人驾驶车辆和电子竞技游戏（例如，雅达利 Atari、Go、扑克和最新的 DOTA2）等。

在基于文本的 NLP 应用领域，机器翻译可能受到深度学习的影响最大。当前，在实际应用中表现最佳的机器翻译系统是基于深度神经网络的模式，例如，谷歌于 2016 年 9 月宣布开发第一阶段的神经网络机器翻译，而微软在 2 个月后发表了类似的声明。Facebook 已经致力于神经网络机器翻译一年左右，到 2017 年 8 月，它正在全面部署。最近，谷歌发布了机器翻译领域最强的 BERT 的多语言模型。BERT 的全称是 Bidirectional Encoder Representations from Transformers，是一种预训练语言表示的最新方法。

BERT 在机器阅读理解顶级水平测试 SQuAD1.1 中表现出惊人的成绩：两个衡量指标上全面超

越人类，而且在 11 种不同 NLP 测试中同样给出了最好的成绩，其中包括将 GLUE 基准推至 80.4%（绝对改进率 7.6%），MultiNLI 准确度达到 86.7%（绝对改进率 5.6%）等。对于机器翻译的解读，本书也会在第 11 章进行深入探讨。

在将深度学习应用于 NLP 问题的过程中，近年来出现的两个重要技术突破是序列到序列学习（Sutskevar 等，2014）和注意力建模（Bahdanau 等，2015）。序列到序列学习引入了一个强大的思想，即利用循环网络以端到端的方式进行编码和解码。虽然注意力建模最初是为了解决对长序列进行编码的困难，但随后的发展显然扩展了它的功能，能够对任意两个序列进行高度灵活的排列，且可以与神经网络参数一起进行学习。与基于统计学习和单词\短语的局部表示的最佳系统相比，序列到序列学习和注意力机制的关键思想提高了基于分布式嵌入的神经网络机器翻译的性能。在这一成功之后不久，这些概念也被成功地应用到其他一些与 NLP 相关的任务中，例如图像字幕生成（Karpathy 和 Fei-Fei，2015；Devlin 等，2015）、语音识别（Chorowski 等，2015）、句法解析、文本理解、问答系统等。

其实，基于神经网络的深度学习模型通常比早期开发的传统机器学习模型更易于设计。在许多应用中，以端到端的方式同时对模型的所有部分执行深度学习，从特征提取一直到预测。促成神经网络模型简化的另一个因素是相同模型构建的模块（例如不同类型的层）通常也可以适用于许多不同的任务。另外，还开发了软件工具包，以便更快更有效地实现这些模型。基于这些原因，深度神经网络现在是大型数据集（包括 NLP 任务）上的各种机器学习和人工智能任务的主要选择方法。

尽管深度学习已经被证明能够以革命性的方式对语音、图像和视频进行重塑处理，并且在许多实际的 NLP 任务中取得了经验上的成功，在将深度学习与基于文本的 NLP 进行交叉时，但其效果却不那么明显。在语音、图像和视频处理中，深度学习通过直接从原始感知数据中学习高级别概念，有效地解决了语义鸿沟问题。然而，在 NLP 中，研究人员在形态学、句法和语义学上提出了更强大的理论和结构化模型，提炼出了理解和生成自然语言的基本机制，但这些机制与神经网络并不容易兼容。与语音、图像和视频信号相比，从文本数据中学习到的神经表征似乎不能同样直接洞察自然语言。因此，将神经网络特别是具有复杂层次结构的神经网络应用于 NLP，近年来得到了越来越多的关注，也已经成为 NLP 和深度学习社区中最活跃的领域，并取得了显著的进步（Deng，2016；Manning 和 Socher，2017）。

1.6　NLP 中深度学习的局限性

目前，尽管深度学习在 NLP 任务中取得了巨大的成功，尤其是在语音识别/理解、语言建模和机器翻译方面，但目前仍然存在着一些巨大的挑战。目前基于神经网络作为黑盒的深度学习方法普遍缺乏可解释性，甚至是远离可解释性，而在 NLP 的理论阶段建立的“理性主义”范式中，专家设计的规则自然是可解释的。在现实工作任务中，其实是迫切需要从“黑盒”模型中得到关于预测的解释，这不仅仅是为了改进模型，也是为了给系统使用者提供有针对性的合理建议（Koh 和 Liang，2017）。

在许多应用中，深度学习方法已经证明其识别准确率接近或超过人类，但与人类相比，它需要更多的训练数据、功耗和计算资源。从整体统计的角度来看，其精确度的结果令人印象深刻，但从

个体角度来看往往不可靠。而且，当前大多数深度学习模型没有推理和解释能力，使得它们容易遭受灾难性失败或攻击，而没有能力预见并因此防止这类失败或攻击。另外，目前的 NLP 模型没有考虑到通过最终的 NLP 系统制定和执行决策目标及计划的必要性。当前 NLP 中基于深度学习方法的一个局限性是理解和推理句子间关系的能力较差，尽管在句子中的词间和短语方面已经取得了巨大进步。

目前，在 NLP 任务中使用深度学习时，虽然我们可以使用基于（双向）LSTM 的标准序列模型，且遇到任务中涉及的信息来自于另外一个数据源时可以使用端到端的方式训练整个模型，但是实际上人类对于自然语言的理解（以文本形式）需要比序列模型更复杂的结构。换句话说，当前 NLP 中基于序列的深度学习系统在利用模块化、结构化记忆和用于句子及更大文本进行递归、树状表示方面还存在优化的空间（Manning，2016）。

为了克服上述挑战并实现 NLP 作为人工智能核心领域的更大突破，有关 NLP 和深度学习研究人员需要在基础研究和应用研究方面做出一些里程碑式的工作。

1.7 NLP 的应用场景

目前，随着自然语言处理领域研究越来越深入，其应用的行业越来越广。比如在文本和语音方面的应用。其中，我们可以看到 NLP 在文本方面的应用有基于自然语言理解的智能搜索引擎和智能检索、智能机器翻译、自动摘要与文本综合、文本分类与文件整理、智能自动作文系统、智能判卷系统、信息过滤与垃圾邮件处理、文学研究与古文研究、语法校对、文本数据挖掘与智能决策以及基于自然语言的计算机程序设计等。在语音方面的应用有机器同声传译、智能远程教学与答疑、语音控制、智能客户服务、机器聊天与智能参谋、智能交通信息服务（ATIS）、智能解说与体育新闻实时解说、语音挖掘与多媒体挖掘、多媒体信息提取与文本转化以及对残疾人智能帮助系统等。下面我们给出一些常见的应用场景。

1. 搜索引擎

在搜索引擎中，我们常常使用词义消歧、指代消解、句法分析等自然语言处理技术，以便更好地为用户提供更加优质的服务。因为我们的搜索引擎不仅仅是为用户提供所寻找的答案，还要做好用户与实体世界连接的贴心服务。搜索引擎最基本的模式就是自动化地聚合足够多的信息，对之进行解析、处理和组织，响应用户的搜索请求并找到对应结果再返回给用户。这里涉及的每一个环节，都需要用到自然语言处理技术。例如，我们日常生活中使用百度搜索“天气”“XX 公交线路”“火车票”等这样略显模糊的需求信息，一般情况下都会得到满意的搜索结果。自然语言处理技术在搜索引擎领域中有了更多的应用，才使得搜索引擎能够快速精准地返回给用户所要的搜索结果。当然，另一方面，正是谷歌和百度这样 IT 巨头商业上的成功，推进了自然语言处理技术的不断进步。

2. 推荐系统

早在 1992 年 Goldberg 就首次给出了一个推荐系统：Tapestry。它其实只是一个个性化的邮件推荐系统，首次提出了协同过滤的思想，利用用户的标注和行为信息对邮件进行重排序。推荐系统

依赖的是数据、算法、人机交互等环节的相互配合，其中使用了数据挖掘、信息检索和计算统计学等技术。我们使用推荐系统的目的是关联用户和一些信息，协助用户找到对其有价值的信息，且让这些信息能够尽快呈现在对其感兴趣的用户面前，从而实现精准推荐。

推荐系统在音乐电影的推荐、电子商务产品推荐、个性化阅读、社交网络好友推荐等场景发挥着重要的作用，美国 Netflix 中 2/3 的电影是因为被推荐而观看的，Google News 利用推荐系统提升了 38%的点击率，Amazon 的销售中推荐占比高达 35%。

3．机器翻译

机器翻译是自然语言处理中最为人知的应用场景，一般是将机器翻译作为某个应用的组成部分，例如跨语言的搜索引流等。目前以 IBM、谷歌、微软为代表的国外科研机构和企业均相继成立机器翻译团队，专门从事智能翻译研究。例如，IBM 于 2009 年 9 月推出 ViaVoiceTranslator 机器翻译软件，为自动化翻译奠定了基础；2011 年开始，伴随着语音识别、机器翻译技术、DNN（深度神经网络）技术的快速发展和经济全球化的需求，口语自动翻译研究已成为当今信息处理领域新的研究热点，Google 于 2011 年 1 月正式在其 Android 系统上推出了升级版的机器翻译服务；微软的 Skype 于 2014 年 12 月宣布推出实时机器翻译的预览版、支持英语和西班牙语的实时翻译，并宣布支持 40 多种语言的文本实时翻译功能。

尤其值得注意的是，在“一带一路”这一发展背景下，合作沟通会涉及 60 多个国家、53 种语言，此时机器翻译的技术应用显得尤为重要，语言的畅通是“一带一路”倡议得以实施的重要基础。机器翻译涉及语义分析、上下文环境等诸多挑战，其发展道路还有很长一段路要走。

4．聊天机器人

聊天机器人是指能通过聊天 App、聊天窗口或语音唤醒 App 进行交流的计算机程序，是被用来解决客户问题的智能数字化助手，其特点是成本低、高效且持续工作。例如，Siri、小娜等对话机器人就是一个应用场景。除此之外，聊天机器人在一些电商网站有着很实用的价值，可以充当客服角色，例如京东客服 JIMI。有很多基本的问题，其实并不需要联系人工客服来解决。通过应用智能问答系统，可以排除掉大量的用户问题，比如商品的质量投诉、商品的基本信息查询等程式化问题，在这些特定的场景中，特别是会被问到高度可预测的问题中，利用聊天机器人可以节省大量的人工成本。图 1-2 给出了一些聊天机器人产品。

5．知识图谱

知识图谱能够描述复杂的关联关系，它的应用极为广泛，最为人所知的就是被用在搜索引擎中丰富搜索结果，并为搜索结果提供结构化结果来体现关联性，这也是谷歌提出知识图谱的初衷。同时微软小冰、苹果 Siri 等聊天机器人中也加入了知识图谱的应用。IBM Watson 是问答系统中应用知识图谱较为典型的例子。按照应用方式，可以将知识图谱的应用分为语义搜索、知识问答以及基于知识的大数据分析和决策等。



图 1-2 部分聊天机器人示意图

语义搜索利用建立大规模知识库对搜索关键词和文档内容进行语义标注，改善搜索结果，如谷歌、百度等在搜索结果中嵌入知识图谱。知识问答是基于知识库的问答，通过对提问句子的语义分析，将其解析为结构化的询问，在已有的知识库中获取答案。在大数据的分析和决策方面，知识图谱起到了辅助作用，典型应用是美国 Netflix 公司利用其订阅用户的注册信息以及观看行为构建的知识图谱反映出英剧版《纸牌屋》很受欢迎，于是拍摄了美剧《纸牌屋》，大受追捧。知识图谱展示如图 1-3 所示。

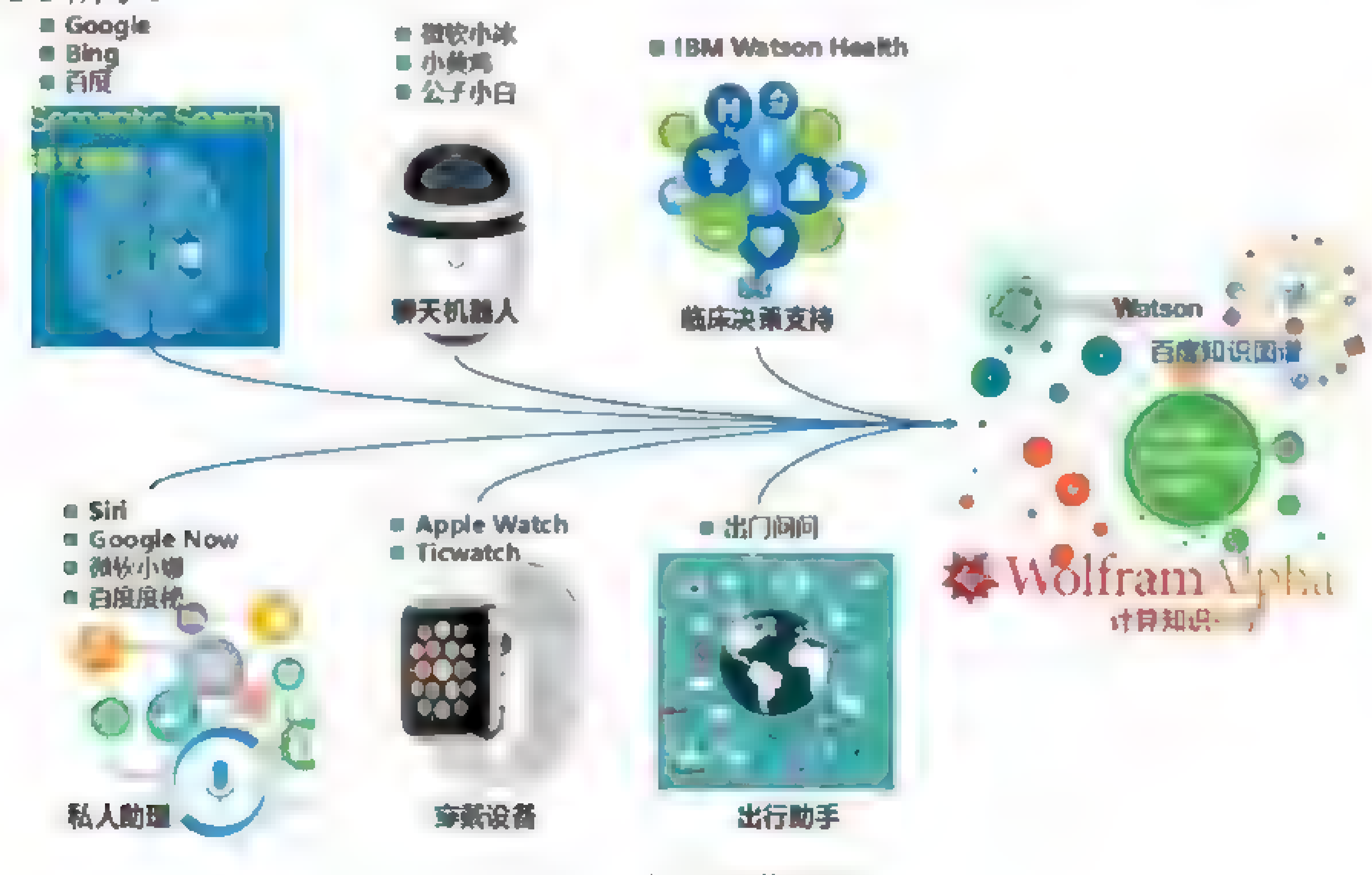


图 1-3 知识图谱展示图

1.8　NLP 的发展前景

随着深度学习时代的来临，神经网络成为一种强大的机器学习工具，并使得自然语言处理取得了许多突破性发展，如情感分析、智能问答、机器翻译等领域都在飞速发展。下面我们梳理一些自然语言处理近期热点和全球热点的情况。

1．文本理解与推理：浅层分析向深度理解迈进

谷歌等公司已经推出了以阅读理解作为深入探索自然语言理解的平台。

文本理解和推理是自然语言处理的重要部分，现在的机器软件已经可以根据文本的上下文来分辨代词等指示词，这是文本理解与推理从浅层分析向深度理解迈进的重要一步。

2．对话机器人：实用化、场景化

从最初 2012 年到 2014 年的语音助手，到 2014 年起逐渐出现的聊天机器人微软小冰、百度小度，再到 2016 年哈尔滨工业大学 SCIR 的笨笨，对话机器人越来越智能。最初的语音助手可以听得到但是听不懂，之后的对话机器人可以听得懂但是实用性却不强，现在对话机器人更多的是和场景结合，即在特定场景做有用的人机对话。

3．NLP+行业：与专业领域深度结合

银行、电器、医药、司法、教育等领域对自然语言处理的需求都非常多。自然语言处理与各行各业的结合越来越紧密，专业化的服务趋势逐渐增强。可以预测，自然语言处理首先会在信息准备充分并且服务方式本身就是知识和信息的领域产生突破，例如医疗、金融、教育和司法领域。

4．学习模式：先验语言知识与深度学习结合

自然语言处理中学习模式有一个较为明显的变化。在浅层到深层的学习模式中，浅层学习是分步骤的，深度学习的方法贯穿在浅层分析的每个步骤中，由各个步骤连接而成。而直接的深度学习则是直接从端到端，人为贡献的知识在深度学习中所占的比重大幅度减小。但如何将深度学习应用于自然语言处理需要进行更多的研究和探索，针对不同任务的不同字词表示，将先验知识和深度学习相结合是未来的一个发展趋势。

5．文本情感分析：事实性文本到情感文本

之前的研究主要是新闻领域的事实性文本，现在情感文本分析更受重视，并且在商业和政府舆情上可以得到很好的应用。例如，2017 年新浪微舆情和哈尔滨工业大学推出“情绪地图”，网民可以登录新浪舆情官方网站查询任何关键词的“情绪地图”，这是语义情绪分析在舆情分析产业上的首次正式应用。

1.9 总结

在本章中，为了建立本书的基本框架，我们首先解释了为什么我们需要 NLP，然后讨论了 NLP 的各类任务。对于 NLP 的来龙去脉，我们又从理性主义和经验主义到当前深层学习浪潮的三次自然语言处理浪潮出发，回顾了自然语言处理领域几十年来的历史发展情况，以便从历史发展中提炼出有助于指导未来方向的见解。接着，我们对于当前 NLP 中深度学习的局限性进行了解读。最后，我们对于 NLP 中的应用场景和前景做了简述。

首先，我们通过一个日常生活中常见的寻找小吃店位置的例子开启了解读 NLP 的篇章，并对 NLP 主要的工作任务进行了初步划分，包括标记化、词义消歧、词性标注、实体命名识别、句子或概要分类、文本生成、问答系统、机器翻译等。

其次，通过对于 NLP 发展的三个阶段的分析，我们知道当前的 NLP 深度学习技术是从前两波发展起来的一种 NLP 技术概念和范式上的革新。这场革新的关键支撑包括通过嵌入对语言实体（子单词、单词、短语、句子、段落、文档等）的分布式表示、嵌入引起的语义概括、语言的大跨度深层序列建模、能够从低到高有效表达语言水平的层次网络以及端到端的深度学习方法，以共同解决许多 NLP 任务中的问题。在深度学习浪潮出现之前，这些都是不可能实现的，这不仅是因为之前的两次发展浪潮缺乏大数据和强大的计算能力，更重要的是在于近年来深度学习范式出现了之前缺少的正确框架。

接着，我们对于当前 NLP 领域深度学习方面的局限性进行了解读，并给出了局限性存在的成因和简要的解决之道。

最后，我们对于 NLP 的常见应用领域进行了说明并给出了 NLP 的发展前景。

总之，深度学习开创了一个新的世界，使得 NLP 比过去任何时候都更具有活力。深度学习不仅提供了一个强大的建模框架，用于表示计算机系统中人类自然语言的认知能力，更重要的是，它已经在 NLP 的许多关键应用领域创造了卓越的实际效果。在本书的其余章节中，将提供使用深度学习框架开发（具体利用 TensorFlow 工具）的 NLP 技术的详细描述，同时也希望本书能够对 NLP 领域的人员有一些帮助，以便使得 NLP 领域有更多突破性成果的出现。接下来的一章，我们将介绍深度学习基础。

第 2 章

深度学习基础

2.1　深度学习介绍

具有机器学习基础的朋友可能都知道，机器学习实际上是一个寻找最优模型的过程。深度学习是机器学习的一个扩展领域，其概念源自于科学家对人工网络长期研究的积累，深度学习的基本结构其实也是深度神经网络。在深度学习下实现的算法集合与人类大脑中的刺激和神经元之间的关系具有相似之处。深度学习在计算机视觉、语言翻译、语音识别、图像识别等方面具有广泛的应用。这些算法集很简单，既可以在有监督的情况下学习，也可以在无监督的情况下学习。

大多数的深度学习算法都是基于人工神经网络的理念，数据丰富，计算资源充足，使得目前世界上对这种算法的训练变得更加容易、深度学习模型的性能得到不断提升。对于这一点，我们可以在图 2-1 中看到更好的表示。

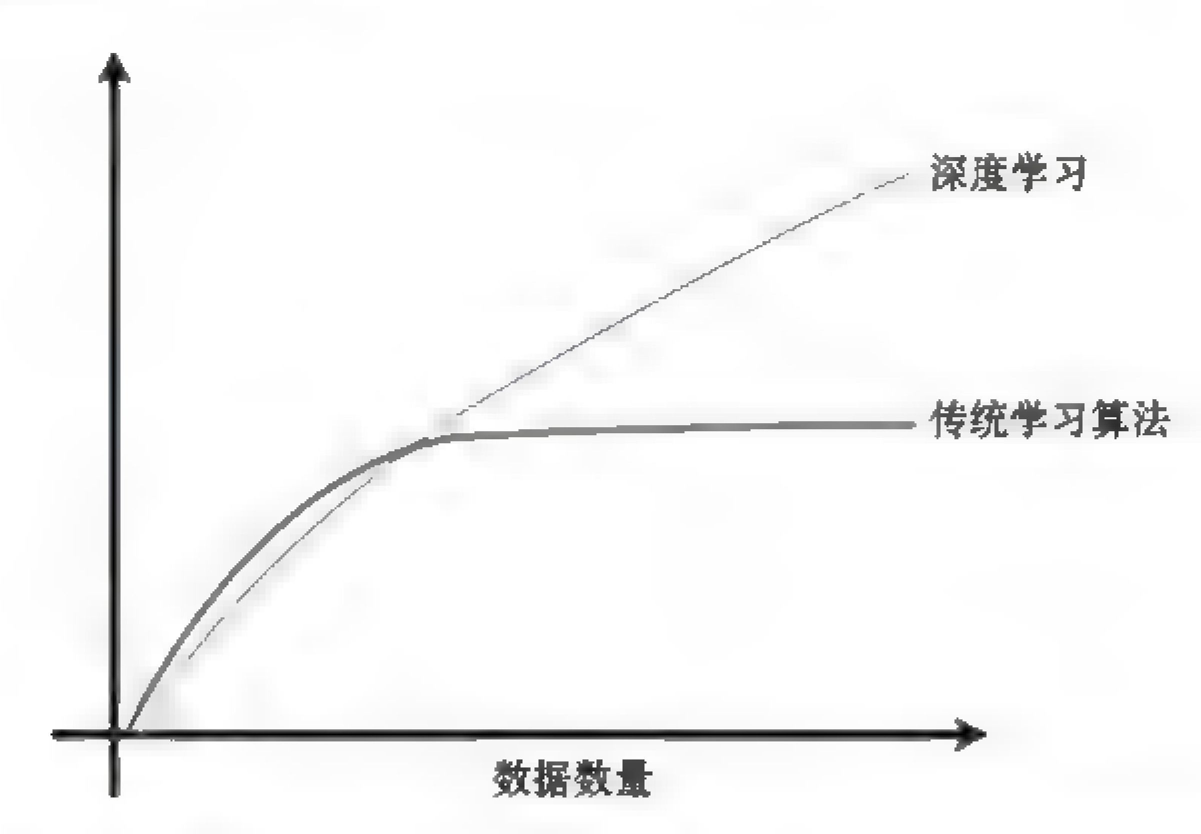


图 2-1　基于神经网络的深度学习算法和传统学习算法效果图

这里深度学习中的“深度”是指人工神经网络结构的深度，而“学习”是指通过人工神经网络

本身进行学习。图 2-2 给出了深度和浅度网络之间差异的展示，以及为什么“深度学习”会受到大家的热捧。

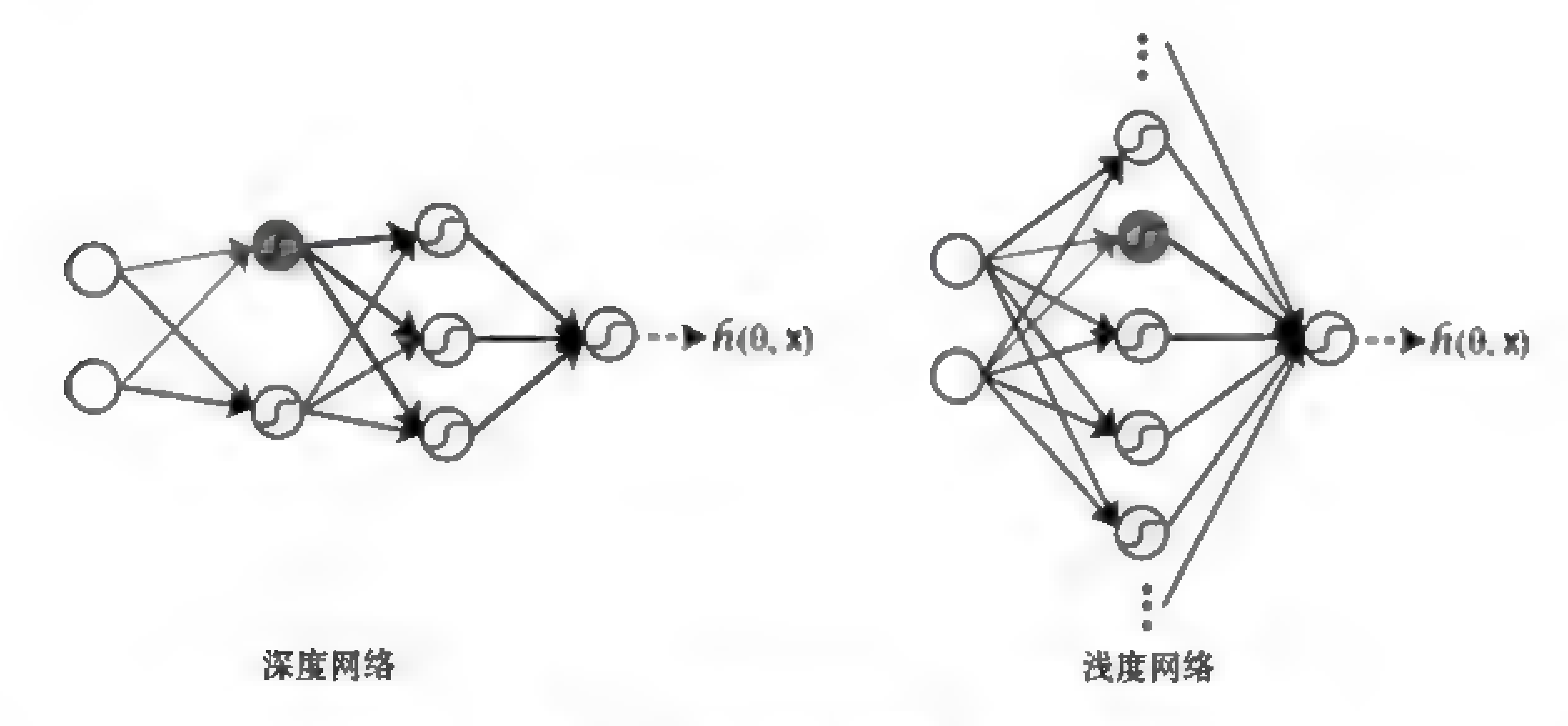


图 2-2 深度和浅度网络示意图

通过利用深度神经网络，我们能够从未标记和非结构化数据（如图像（像素数据）、文档（文本数据）或文件（音频、视频数据））中寻找出潜在的结构情况（或特征学习）。实际上，虽然深度学习中的人工神经网络和模型在本质上具有相似的结构，但这并不意味着两个人工神经网络的组合在利用数据进行训练时，我们获得的表现（或效果）与深度神经网络就相似。其中一个重要的原因是，深度神经网络与普通人工神经网络之间所使用的反向传播方式不同。

接下来，我们先介绍一下深度学习的演变过程，然后从神经网络开始详细解读。

2.2 深度学习演变简述

2.2.1 深度学习早期

1943 年，心理学家麦卡洛克和数学逻辑学家皮兹发表了论文《神经活动中内在思想的逻辑演算》，提出了 MP 模型。MP 模型是模仿神经元的结构和工作原理构建的一个基于神经网络的数学模型，本质上是一种“模拟人类大脑”的神经元模型。MP 模型作为人工神经网络的起源，开创了人工神经网络的新时代，也奠定了神经网络模型的基础。

1949 年，加拿大著名心理学家唐纳德·海布在《行为的组织》中提出了一种基于无监督学习的规则——海布学习规则（Hebb Rule）。海布学习规则模仿人类认知世界的过程建立一种“网络模型”，该网络模型针对训练数据集进行大量的训练并提取训练集的统计特征，然后按照样本的相似程度进行分类，把相互之间联系密切的样本分为一类，这样就把样本分成了若干类。海布学习规则与“条件反射”机理一致，为后面的神经网络学习算法奠定了基础，具有重大的历史意义。

20 世纪 50 年代末，在 MP 模型和海布学习规则的研究基础上，美国科学家罗森布拉特发现了一种类似于人类学习过程的学习算法——感知机学习，并于 1958 年正式提出了由两层神经元组成

的神经网络，称之为“感知器”。感知器本质上是一种线性模型，可以对输入的训练集数据进行二分类，且能够在训练集中自动更新权重值。感知器的提出吸引了大量科学家对人工神经网络研究的兴趣，对神经网络的发展具有里程碑式的意义。

随着研究的深入，在 1969 年，“AI 之父”马文·明斯基和 LOGO 语言的创始人西蒙·派珀特共同编写了一本图书《感知器》，在书中他们证明了单层感知器无法解决线性不可分问题（例如异或问题）。由于这个致命的缺陷以及没有及时推广感知器到多层神经网络中，在 20 世纪 70 年代人工神经网络进入了第一个寒冬期，人们对神经网络的研究也停滞了将近 20 年。

2.2.2　深度学习的发展

1982 年，著名物理学家约翰·霍普菲尔德发明了 Hopfield 神经网络。Hopfield 神经网络是一种结合存储系统和二元系统的循环神经网络。Hopfield 网络也可以模拟人类的记忆，根据激活函数的选取不同，有连续型和离散型两种类型，分别用于优化计算和联想记忆。由于容易陷入局部最小值的缺陷，该算法并未在当时引起很大的轰动。

直到 1986 年，深度学习之父杰弗里·辛顿提出了一种适用于多层感知器的反向传播算法——BP 算法。BP 算法在传统神经网络正向传播的基础上，增加了误差的反向传播过程。反向传播过程不断地调整神经元之间的权重值和阈值，直到输出的误差减小到允许的范围之内，或达到预先设定的训练次数为止。BP 算法完美地解决了非线性分类问题，让人工神经网络再次引起了人们广泛的关注。

20 世纪 80 年代计算机的硬件水平有限，比如运算能力跟不上，导致当神经网络的规模增大时使用 BP 算法出现“梯度消失”的问题。这使得 BP 算法的发展受到了很大的限制。再加上 20 世纪 90 年代中期，以 SVM 为代表的其他浅层机器学习算法的提出，及其在分类、回归问题上均取得了很好的效果，其原理又明显不同于神经网络模型，所以人工神经网络的发展再次进入了瓶颈期。

2.2.3　深度学习的爆发

2006 年，杰弗里·辛顿以及他的学生鲁斯兰·萨拉赫丁诺夫正式提出了深度学习的概念。他们在世界顶级学术期刊《科学》发表的一篇文章中详细地给出了“梯度消失”问题的解决方案通过无监督的学习方法逐层训练算法，再使用有监督的反向传播算法进行调优。该深度学习方法的提出，立即在学术圈引起了巨大的反响，以斯坦福大学、多伦多大学为代表的众多世界知名高校纷纷投入巨大的人力、财力进行深度学习领域的相关研究，而后又迅速蔓延到工业界中。

2012 年，在著名的 ImageNet 图像识别大赛中，杰弗里·辛顿领导的小组采用深度学习模型 AlexNet 一举夺冠。AlexNet 采用 ReLU 激活函数，从根本上解决了梯度消失问题，并采用 GPU 极大地提高了模型的运算速度。同年，由斯坦福大学的吴恩达教授和世界顶尖计算机专家 Jeff Dean 共同主导的深度神经网络——DNN 技术在图像识别领域取得了惊人的成绩，在 ImageNet 评测中成功地把错误率从 26%降低到了 15%。深度学习算法在世界大赛中脱颖而出，再一次吸引了学术界和工业界对深度学习领域的关注。

随着深度学习技术的不断进步以及数据处理能力的不断提升，2014 年，Facebook 基于深度学习技术的 DeepFace 项目，在人脸识别方面的准确率已经能达到 97%以上，跟人类识别的准确率几乎没有差别。这样的结果也再一次证明了深度学习算法在图像识别方面的一骑绝尘。

2016 年，随着谷歌公司基于深度学习开发的 AlphaGo 以 4:1 的比分战胜了国际顶尖围棋高手李世石，深度学习的热度一时无两。后来，AlphaGo 又接连和众多世界级围棋高手过招，均取得了完胜。这也证明了在围棋界，基于深度学习技术的机器人已经超越了人类。

2017 年，基于强化学习算法的 AlphaGo 升级版 AlphaGo Zero 横空出世。其采用“从零开始”“无师自通”的学习模式，以 100:0 的比分轻而易举打败了之前的 AlphaGo。除了围棋，它还精通国际象棋等其他棋类游戏，可以说是真正的棋类“天才”。此外在这一年，深度学习的相关算法在医疗、金融、艺术、无人驾驶等多个领域均取得了显著的成果。所以，也有专家把 2017 年看作是深度学习甚至是人工智能发展最为突飞猛进的一年。

2.3 神经网络介绍

神经网络这个词，其实是一个非常泛的概念，有两个类别：生物神经网络和人工神经网络。生物神经网络是研究生物学的，一般是指生物的大脑神经元、细胞、触点等组成的网络，用于产生生物的意识、帮助生物进行思考和行动。人工神经网络（Artificial Neural Networks，ANN）有时也称为神经网络（NN）或连接模型（Connection Model），它是一种模仿动物神经网络的行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的。人工神经网络是一种应用类似于大脑神经突触联接的结构进行信息处理的数学模型。在工程与学术界也常直接简称为“神经网络”或类神经网络。所以，我们有必要先简要介绍一下生物神经网络中的神经元模型。

神经元

对于神经元的研究由来已久，1904 年生物学家就已经知晓了神经元的组成结构。一个神经元通常具有多个树突，主要用来接受传入信息；而轴突只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息。轴突末梢跟其他神经元的树突产生连接，从而传递信号。这个连接的位置在生物学上叫作“突触”。神经元的基本功能是通过接受、整合、传导和输出信息实现信息交换。这样一来，神经元按照用途可以分为三种：输入神经、传出神经和连体神经。人脑中的神经元形状如图 2-3 所示。

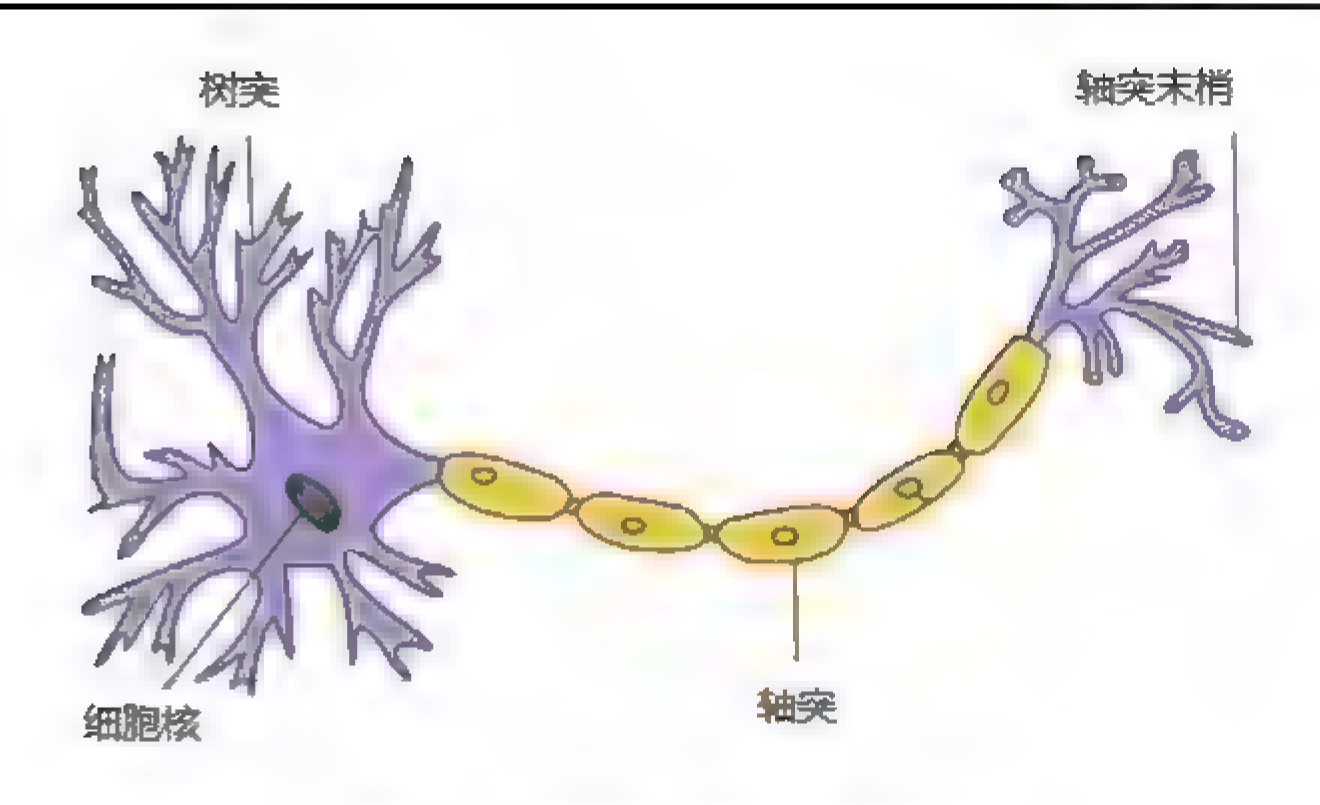


图 2-3　人脑中的神经元形状图

2.4　神经网络的基本结构

神经网络背后的原理源自于一些基本元素组成的集合，如人工神经元或感知器（Perceptron），这些元素最早是由弗兰克•罗森布拉特（Frank Rosenblatt）在 20 世纪 50 年代开发的。它们有几个二进制输入（0 或 1），$x_1,x_2,\ldots,x_n$，如果这些输入的总和大于激活电位（等同于偏差），则产生单个二进制的输出。每当超过激活电位时，神经元被称为“激活”，并表现为一个阶跃函数（step function）。发射信号的神经元将信号传递给与其树突相连的其他神经元，如果超过激活电位，树突就会激活信号，从而产生级联效应，如图 2-4 所示，其实这也是早期的单层神经网络（感知器）。

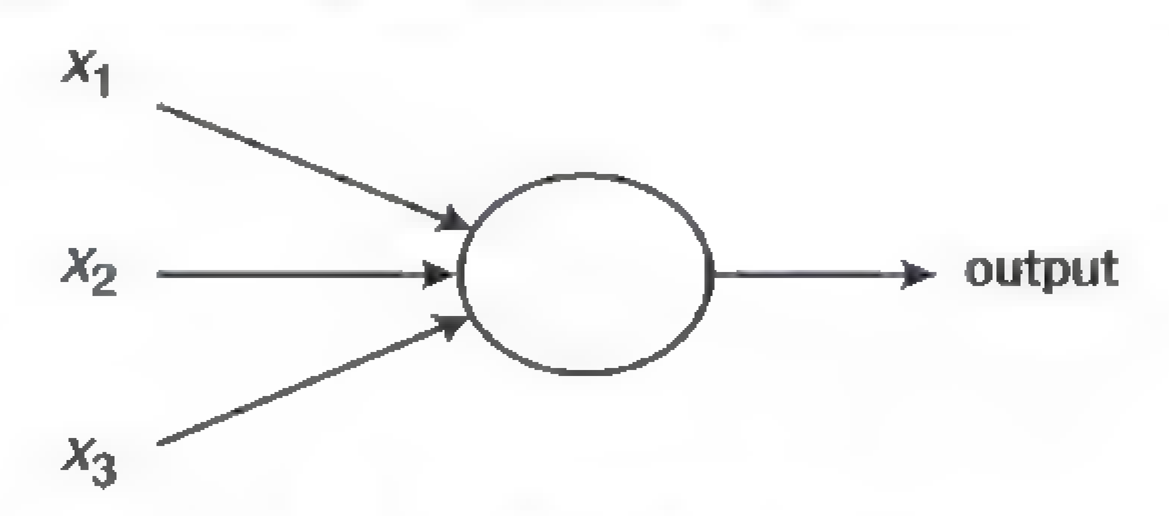


图 2-4　神经元样例示意图

由于并非所有输入都具有相同的重要性，因此每个输入 x_i 都被分配了自己的权重值，允许模型为某些输入指定更高的权重值。因此，若加权和大于激活电位或偏差，则输出为 1，即可以给出输出的结果，具体如下：

$$output = \sum_j w_j x_j + Bias \tag{2.1}$$

实际上，阶跃函数的跳跃性质使得它具有不连续、不光滑、不可导等特点，如果利用这种简单形式的函数，我们很难达到理想的效果，如图 2-5 所示。因此，在实际应用中，我们通常不会直接采用阶跃函数来作为激活函数，我们需要对其进行两处修改，以使其表现得更可预测，即权重值和偏差的微小变化仅导致输出的微小变化，这两处具体如下：

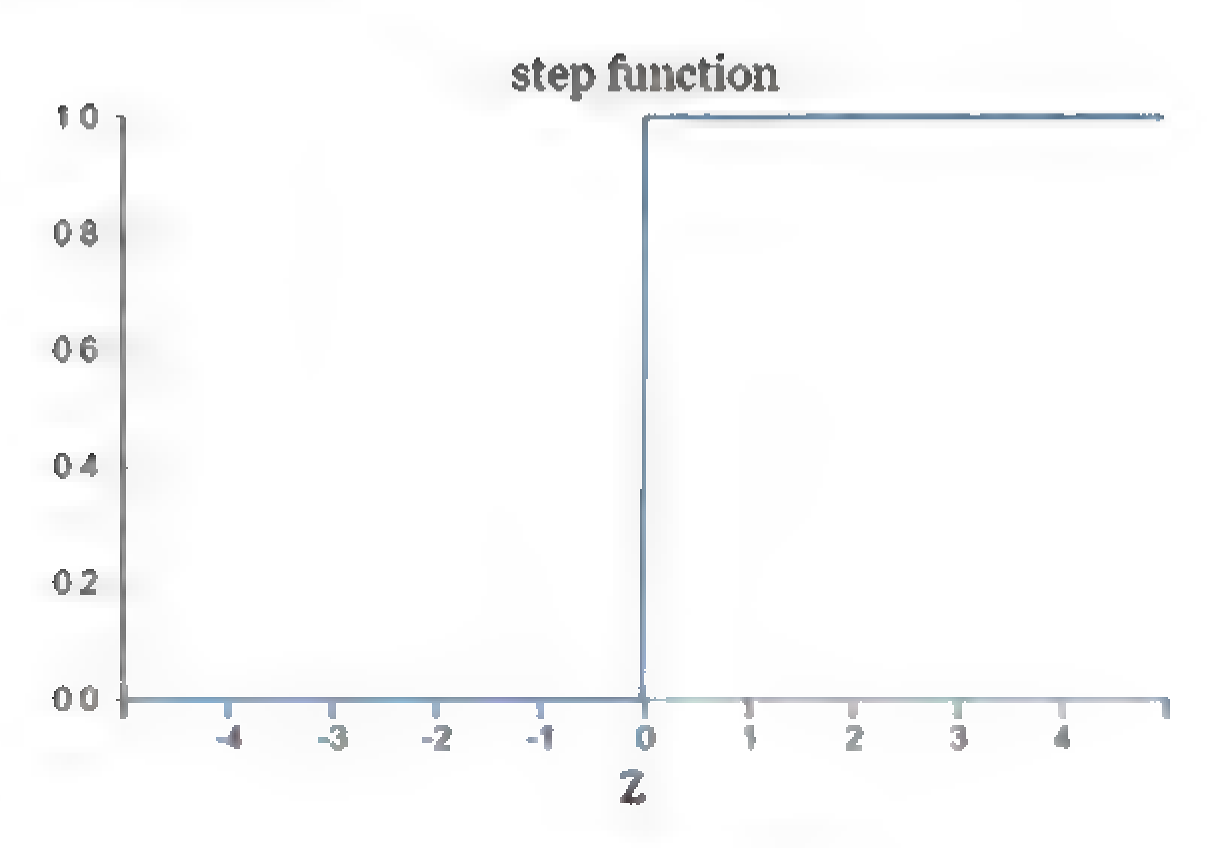


图 2-5 阶跃函数示意图

（1）输入可以取 0~1 之间的任何值，而不是二进制（0 或 1）；

（2）为了使输出在给定的输入$x_1, x_2, \dots, x_N$、权重值$w_1, w_2, \dots, w_N$和偏差b下更平稳地工作，我们使用以下 Sigmoid 函数（见图 2-6）：

$$\sigma(x_i, w_i) = \frac{1}{1+\exp(-\sum_j x_j w_j - b)} \tag{2.2}$$

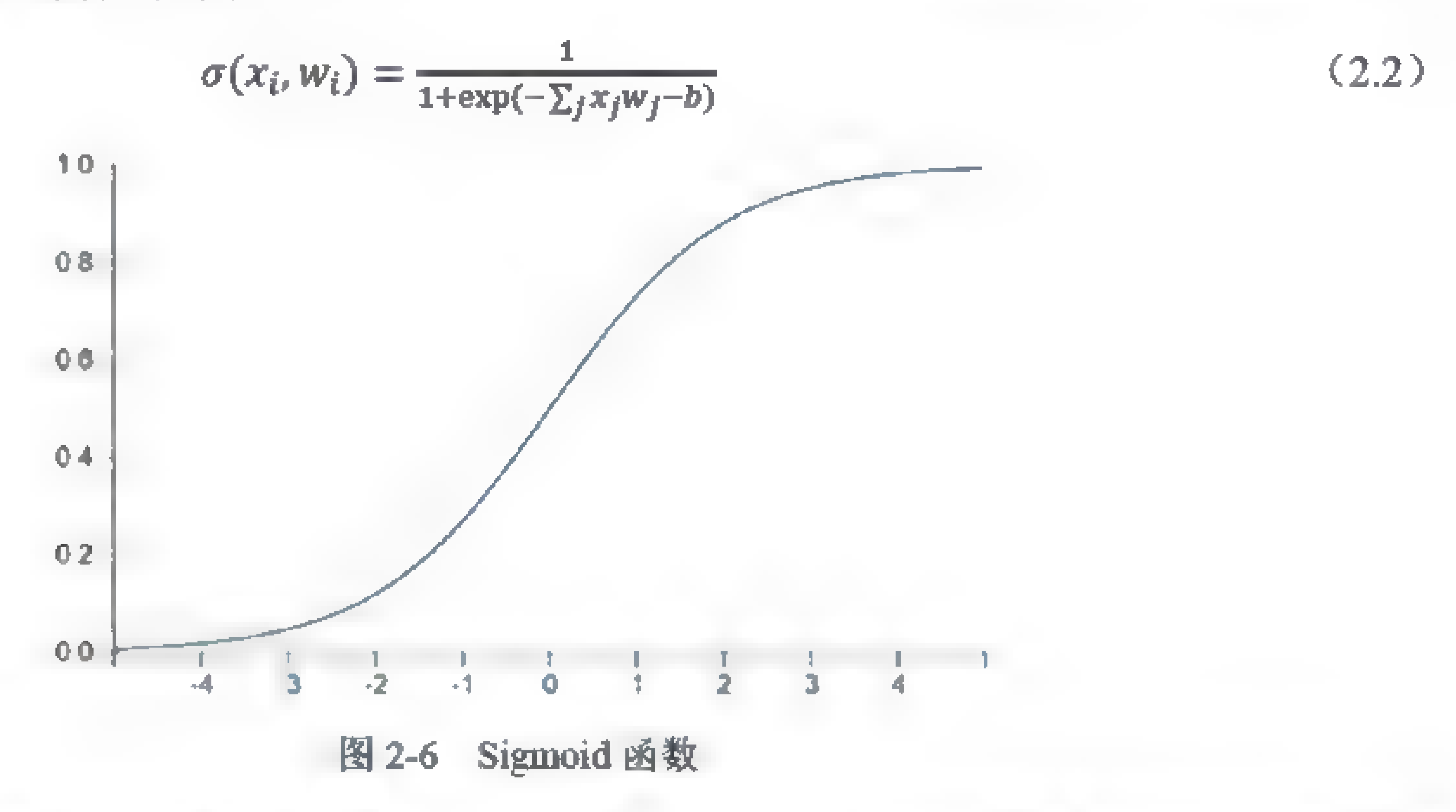


图 2-6 Sigmoid 函数

指数函数（或σ）的平滑度意味着权重值和偏差的微小变化将引起神经元输出的微小变化（变化可能是权重值和偏差变化的线性函数）。

除了通常的 Sigmoid 函数外，更常用的其他非线性函数还包括以下几个函数，它们中的每一个都可能具有类似或不同的输出范围，因此可以相应地使用。

ReLU：线性整流函数（Rectified Linear Unit，ReLU），又称修正线性单元，是一种人工神经网络中常用的激活函数（Activation Function），通常指代以斜坡函数及其变种为代表的非线性函数。这将使激活保持在 0 位，使用以下函数计算：

$$Z_j = f_j(x_j) = \max(0, x_j) \tag{2.3}$$

其中，x_j是第j个输入值，z_j是经过 ReLU 函数f处理过的相应输出值。ReLU 函数的图形如图 2-7 所示，对于所有 x≤0，函数值为 0；对于所有 x>0，函数线性斜率为 1。

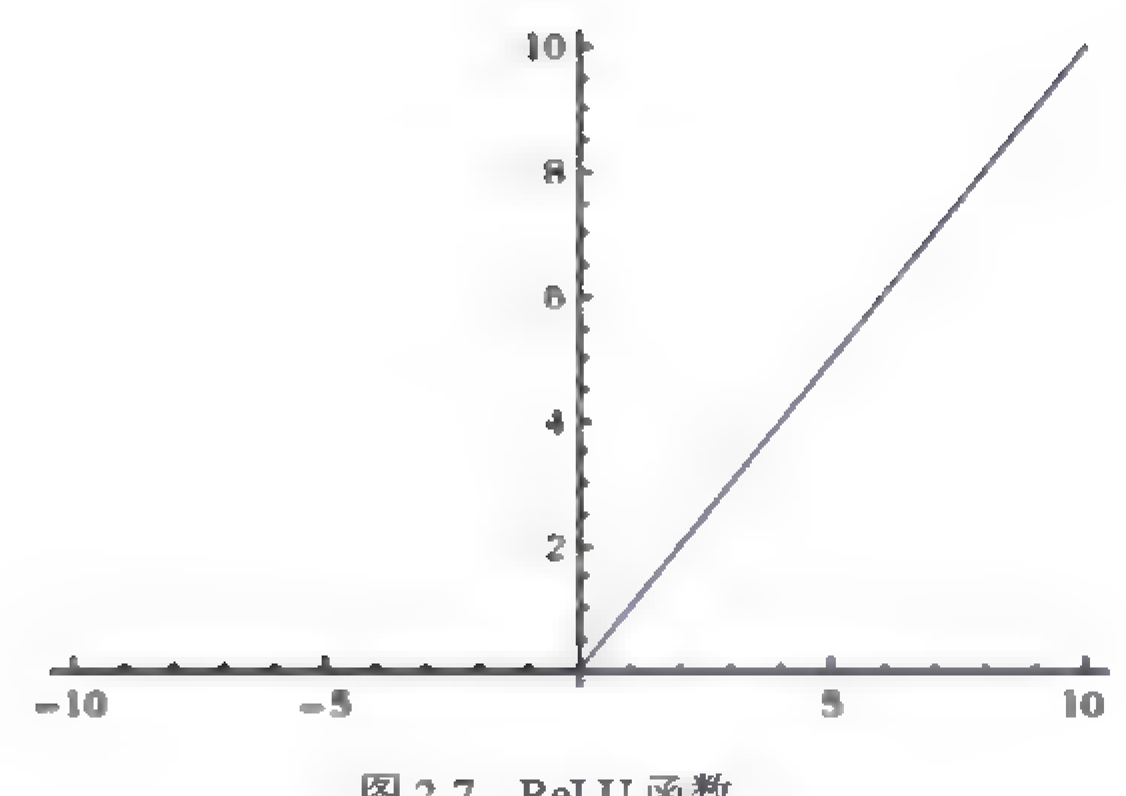


图 2-7 ReLU 函数

ReLU 经常面临着“死亡”的问题，特别是当学习率被设置为更高的数值时，因为这会触发不允许激活特定神经元的权重值更新，从而使该神经元的梯度永远为零。 ReLU 提供的另一个风险是激活函数的爆炸，因为输入值x_j本身就是输出。ReLU 也存在不少优点，例如在x_j低于 0 的情况下引入稀疏性，引起稀疏表示，并且当 ReLU 不变时返回梯度，它会导致更快的学习，伴随着梯度消失的可能性会降低。

下面给出其他几个常见的函数，这些函数能够使我们很容易对梯度下降进行训练，具体如下：

- LReLU（Leaky ReLU）：通过为 x 小于 0 的值引入略微减小的斜率（约 0.01）来减轻 ReLU 死亡的问题，LReLU 确实提供了很多成功的场景，尽管有时也会出错。
- ELU（指数线性单位）：它们提供负值，通过将附近的梯度移动到单位自然梯度，将平均单位激活推至接近零，从而加快学习过程。有关 ELU 更详细的解释，请参阅 Djork-ArnéClevert 的原始论文，网址为 https://arxiv.org/abs/1511.07289。
- softmax：也被称为归一化指数函数，它转换（0,1）范围内的一组给定实值，使得组合和为 1。softmax 函数表示如下：

$$\sigma(z)_j = e^{zk} / \sum_{k=1}^{k} e^{zk} \quad (2.4)$$

这里，$j = 1,2,\ldots,K$。

与哺乳动物大脑一样，单个神经元也是分层组织的，在一层内与下一层相连，形成一个 ANN 或者说人工神经网络或多层感知器（MLP）。这样，整个网络的复杂性就是基于这些元素和连接相邻层的数量情况。

输入和输出之间的层称为隐藏层，层之间的连接密度和类型是结构（也叫配置）。例如，一个完全连接的结构将 L 层的所有神经元连接到 L+1 层的所有神经元。若要具有更明显的结构，我们只能将一个局部邻域连接到下一层。图 2-8 显示了两个具有密集连接的隐藏层。

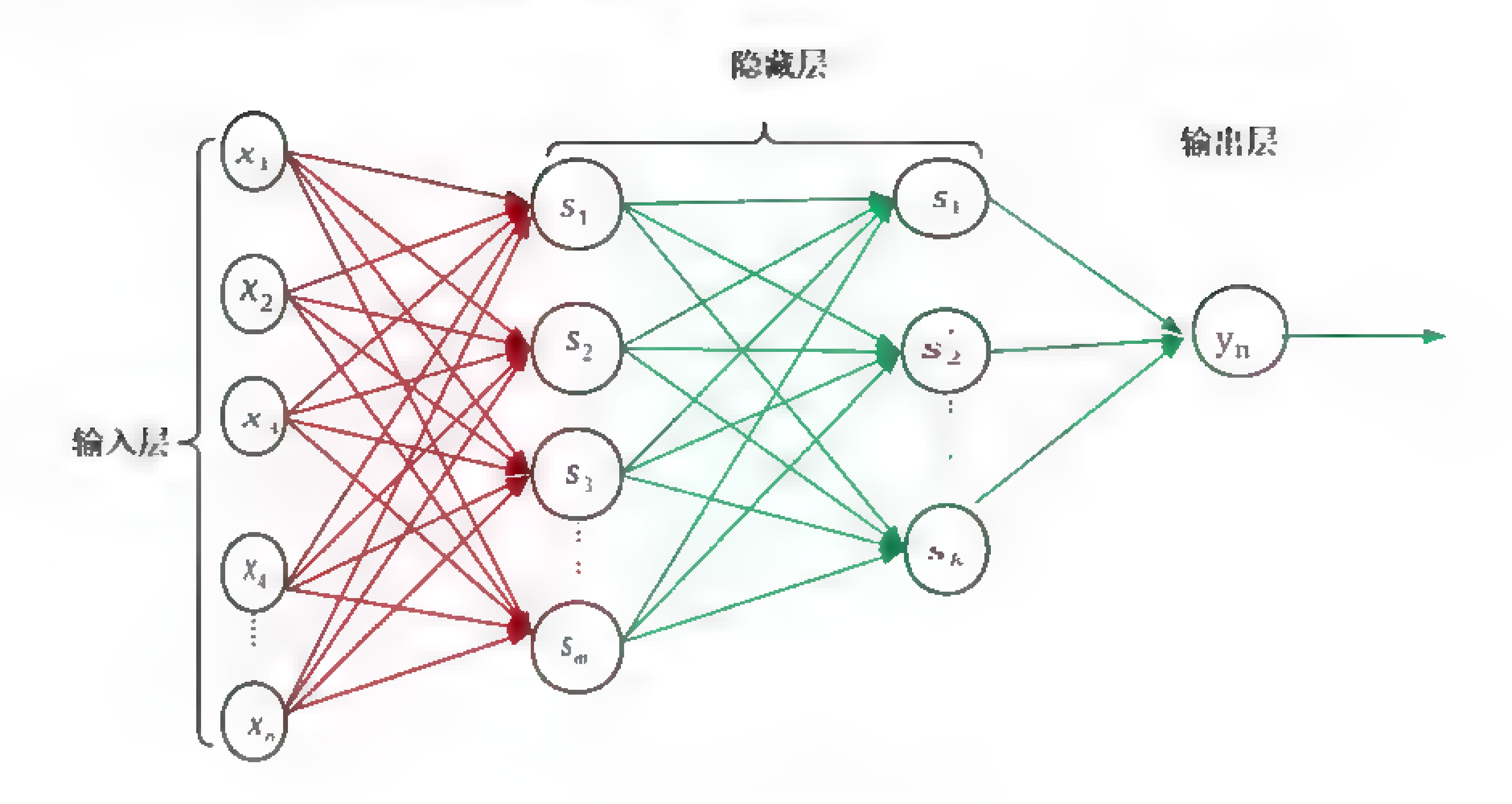


图 2-8 神经网络架构示例图

2.5 两层神经网络（多层感知器）

2.5.1 简述

两层神经网络是本文的重点，因为正是从两层结构为起点神经网络才开始了大范围的推广与使用。

大家知道，单层神经网络无法解决异或问题，但是当增加一个计算层以后，两层神经网络不仅可以解决异或问题，而且具有非常好的非线性分类效果。不过，两层神经网络的计算则是一个问题，因为没有一个较好的解法。

1986 年，Rumelhar 和 Hinton 等人提出了反向传播（Backpropagation，BP）算法，解决了两层神经网络所需要的复杂计算量的问题，从而带动了业界使用两层神经网络研究的热潮。目前，大量讲解神经网络的教材之内容基本都是以介绍两层（带一个隐藏层）神经网络为重点。

当时的 Hinton 还很年轻，30 年以后，正是他重新定义了神经网络，带来了神经网络复苏的又一波发展浪潮。

2.5.2 两层神经网络结构

两层神经网络除了包含一个输入层和一个输出层以外，还增加了一个中间层。此时，中间层和输出层都是计算层。我们扩展上节的单层神经网络，在右边新加一个层次。

现在，我们的权重值矩阵增加到了两个，我们用上标来区分不同层次之间的变量。

例如，a_x^y代表第y层中的第x个节点。z_1, z_2变成了a_1^2, a_2^2。图 2-9 给出了a_1^2, a_2^2的计算公式。

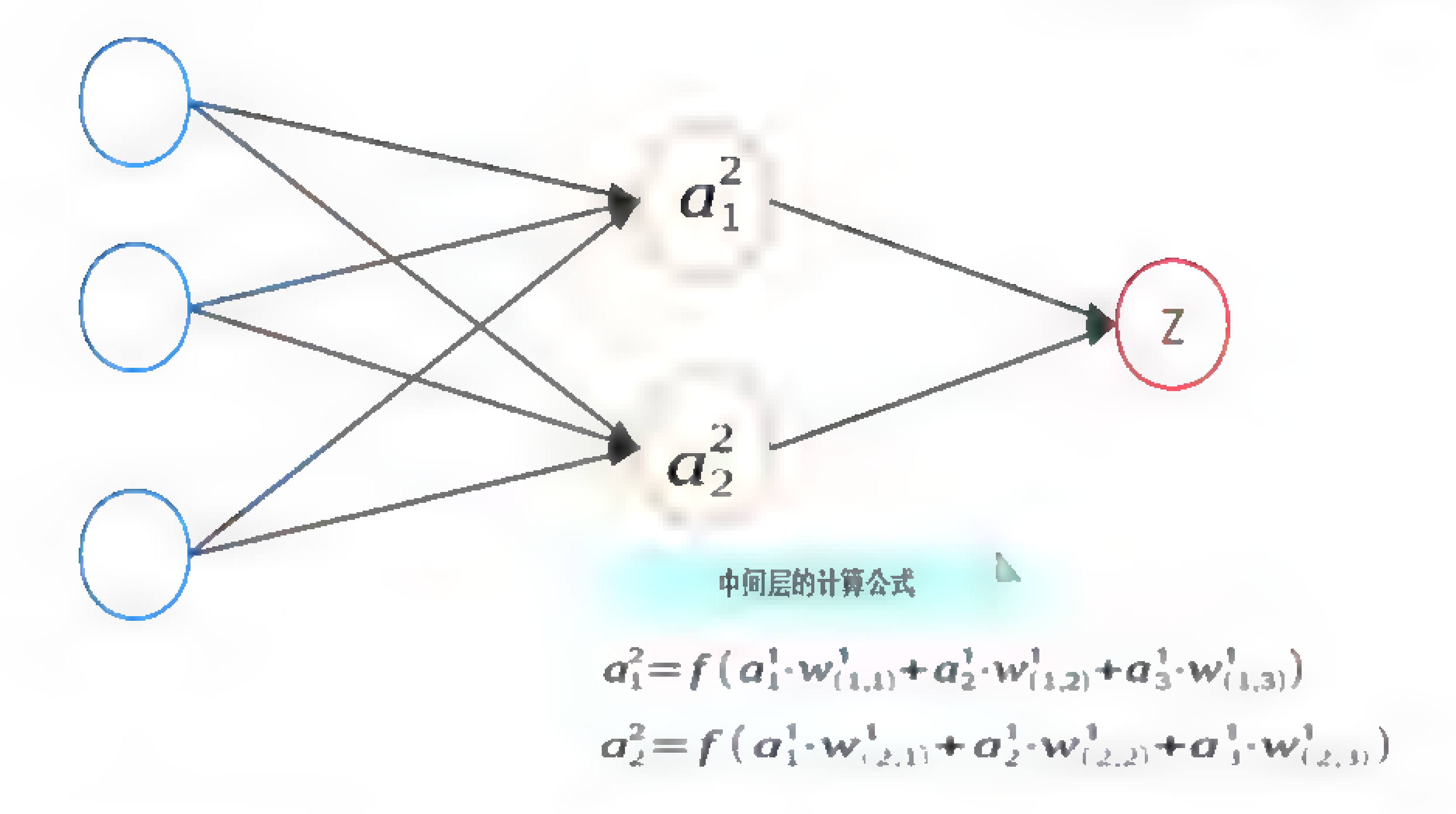


图 2-9　两层神经网络（中间层计算）

计算最终输出 z 的方式是利用了中间层的a_1^2, a_2^2和第二个权值矩阵计算得到的，如图 2-10 所示。

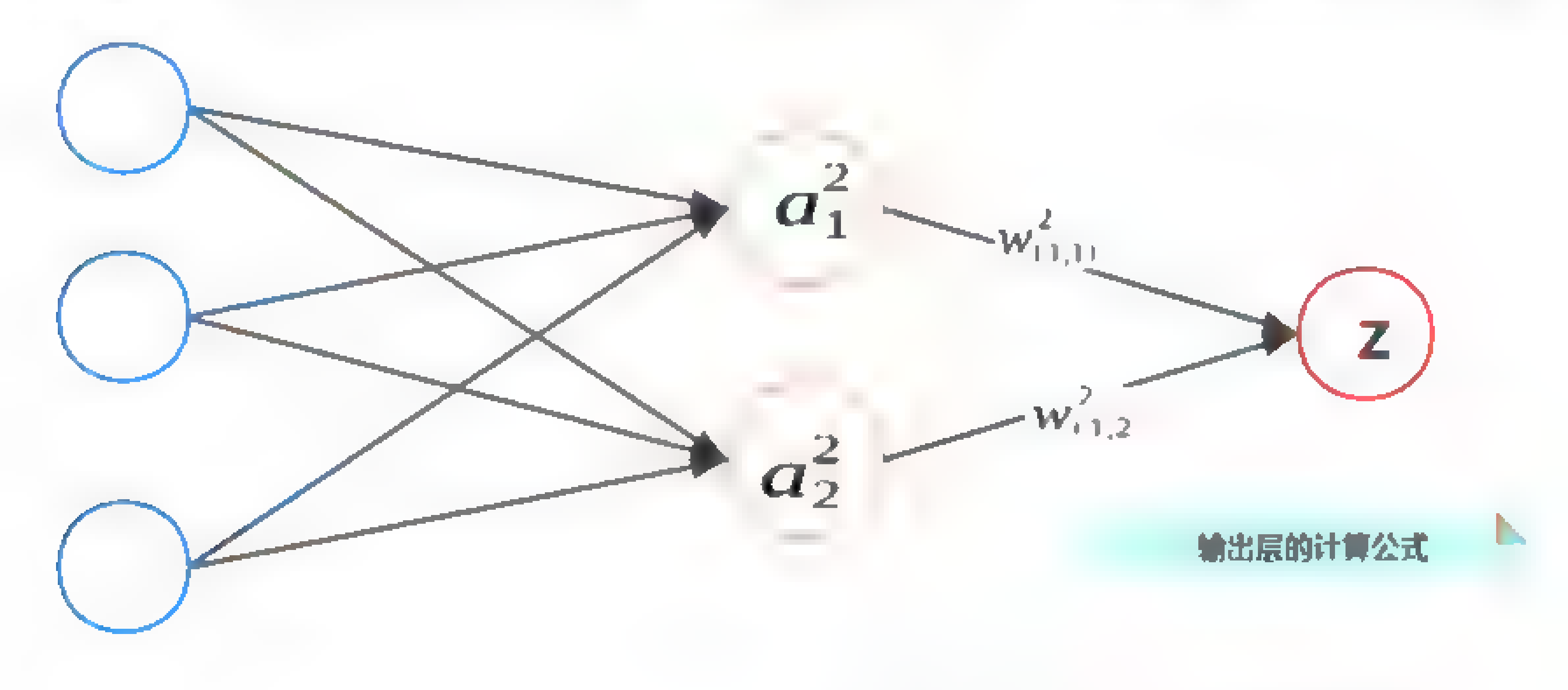


图 2-10　两层神经网络（输出层计算）

假设我们的预测目标是一个向量，那么与前面类似，只需要在“输出层”再增加节点即可。

2.6　多层神经网络（深度学习）

2.6.1　简述

人工神经网络在被人摒弃的 10 年中，有几个学者仍然在坚持研究，其中的典型代表就是加拿大多伦多大学的 Geoffery Hinton 教授。

2006 年，Hinton 在《*Science*》和相关期刊上发表了论文，首次提出了“深度信念网络”的概

念。与传统的训练方式不同，"深度信念网络"有一个"预训练"（Pre-Training）的过程，可以方便地让神经网络中的权重值找到一个接近最优解的值，之后再使用"微调"（Fine-Tuning）技术来对整个网络进行优化训练。这两个技术的运用大幅度减少了训练多层神经网络的时间。他给多层神经网络相关的学习方法赋予了一个新名词——"深度学习"。

很快，深度学习在语音识别领域崭露头角。接着，2012 年，深度学习技术又在图像识别领域大展拳脚。Hinton 与他的学生在 ImageNet 竞赛中用多层的卷积神经网络成功地对包含一千类别的一百万张图片进行了训练，取得了分类错误率 15%的好成绩，这个成绩比第二名高了近 11 个百分点，这充分证明了多层神经网络识别效果的优越性。

在这之后，关于深度神经网络的研究与应用不断涌现。

关于 CNN（Conventional Neural Network，卷积神经网络）与 RNN（Recurrent Neural Network，循环神经网络）及其变体 LSTM 的架构，我们会在后面的章节中进行单独解读，这里不做过多阐释。

2.6.2 多层神经网络结构

我们延续两层神经网络的方式来设计一个多层神经网络。

在两层神经网络的输出层后面，继续添加层次。原来的输出层变成中间层，新加的层次成为新的输出层，可以得到图 2-11。

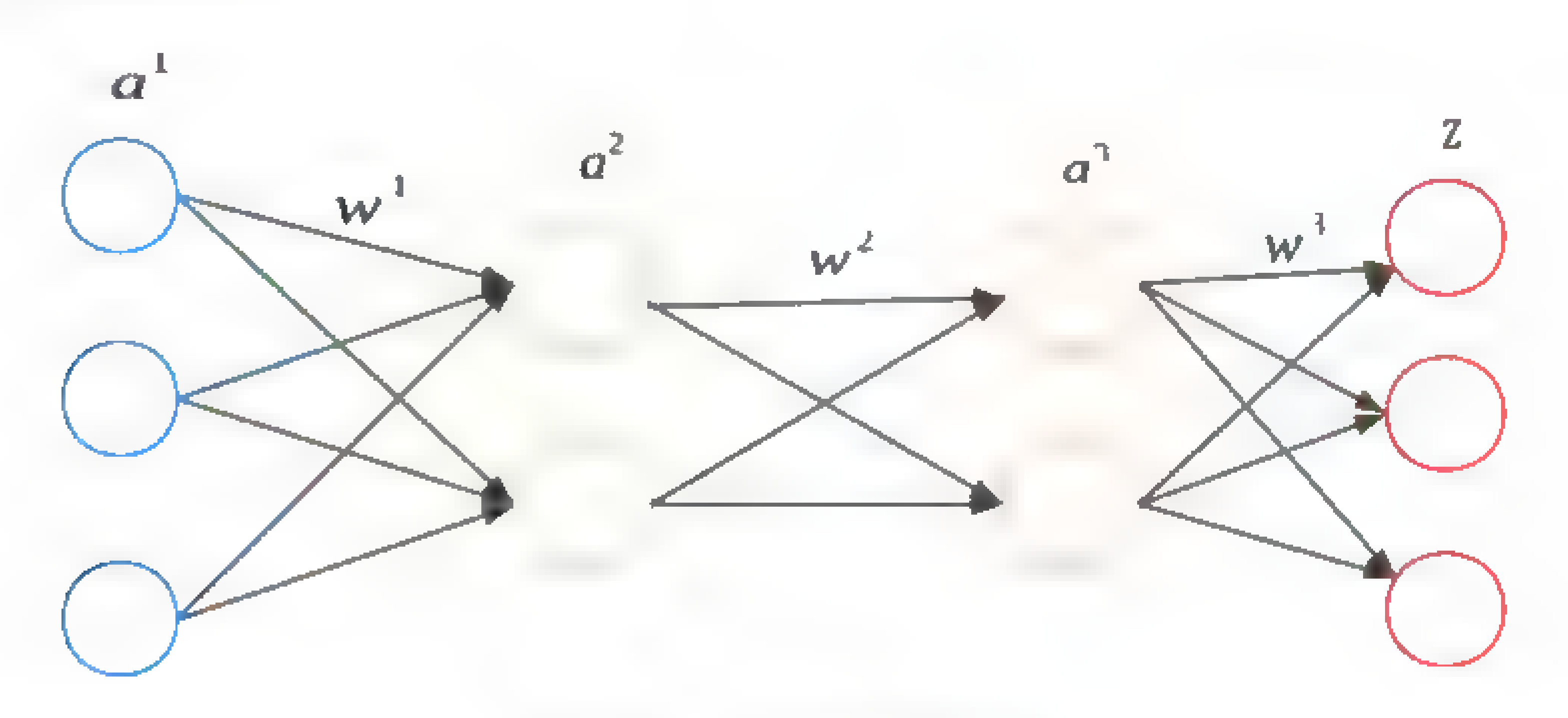


图 2-11 多层神经网络

依照这样的方式不断添加，我们可以得到更多层的多层神经网络。公式推导的话其实跟两层神经网络类似，使用矩阵运算的话就仅仅是加一个公式而已。

2.7 编码器-解码器网络

编码器-解码器（Encoder-Decoder）网络使用一个网络来创建输入的内部表示，或者对其进行"编码"，并且该表示用作另一个网络的输入以产生输出。这有助于超越输入的分类。最终输出可以是相同的模态，即语言翻译，或基于概念的不同模态，例如图像的文本标记。作为参考，可以阅

读谷歌团队发表的论文“使用神经网络进行序列学习”（https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf）。

编码器-解码器网络其实可以看作一个解决问题的框架，主要解决 seq2seq 类问题，Sequence（序列）在这里可以理解为一个字符串序列，当我们在给定一个字符串序列后，希望得到与之对应的另一个字符串序列，比如问答系统、翻译系统。

编码器-解码器网络的流程可以理解为“编码—存储—解码”这一流程，可以用人脑流程来类比。我们先看到源 Sequence，将其读一遍，然后在我们大脑当中就记住了这个源 Sequence，并且存在大脑的某一个位置上，形成我们自己的记忆（对应后面的 Context），然后我们再经过思考，将这个大脑里的东西转变成输出，写下来。我们大脑读入的过程叫作编码器（Encoder），即将输入的东西变成我们自己的记忆，放在大脑当中，而这个记忆可以叫作 Context，然后我们再根据这个 Context 转化成答案写下来，这个写的过程叫作解码器（Decoder）。

整个模型可以用图 2-12 表示。

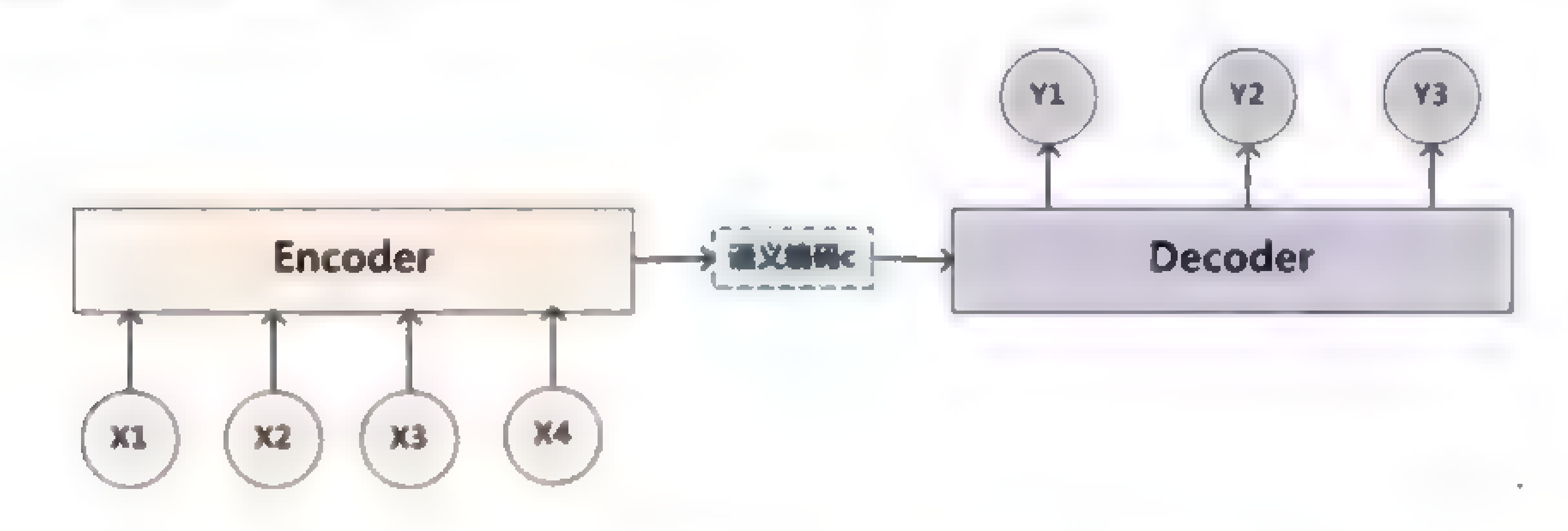


图 2-12　编码器-解码器（Encoder-Decoder）网络架构

2.8　随机梯度下降

几乎所有优化问题的解决方案都是梯度下降算法。它是一种迭代算法，通过随后更新函数的参数来最小化损失函数。

从图 2-13 中可以看到，我们首先把函数想象成一种山谷，想象一个球滚下山谷斜坡的情形。日常生活经验告诉我们，球最终会滚到谷底。也许我们可以用这个方法求出损失函数的最小值。

我们这里使用的函数依赖于两个变量：v1 和 v2。这可能是显而易见的，因为我们的损失函数看起来像前面的那个。为了达到这样一个平滑的损失函数，我们取二次损失：

$$(y - y^{\text{predicted}})^2$$

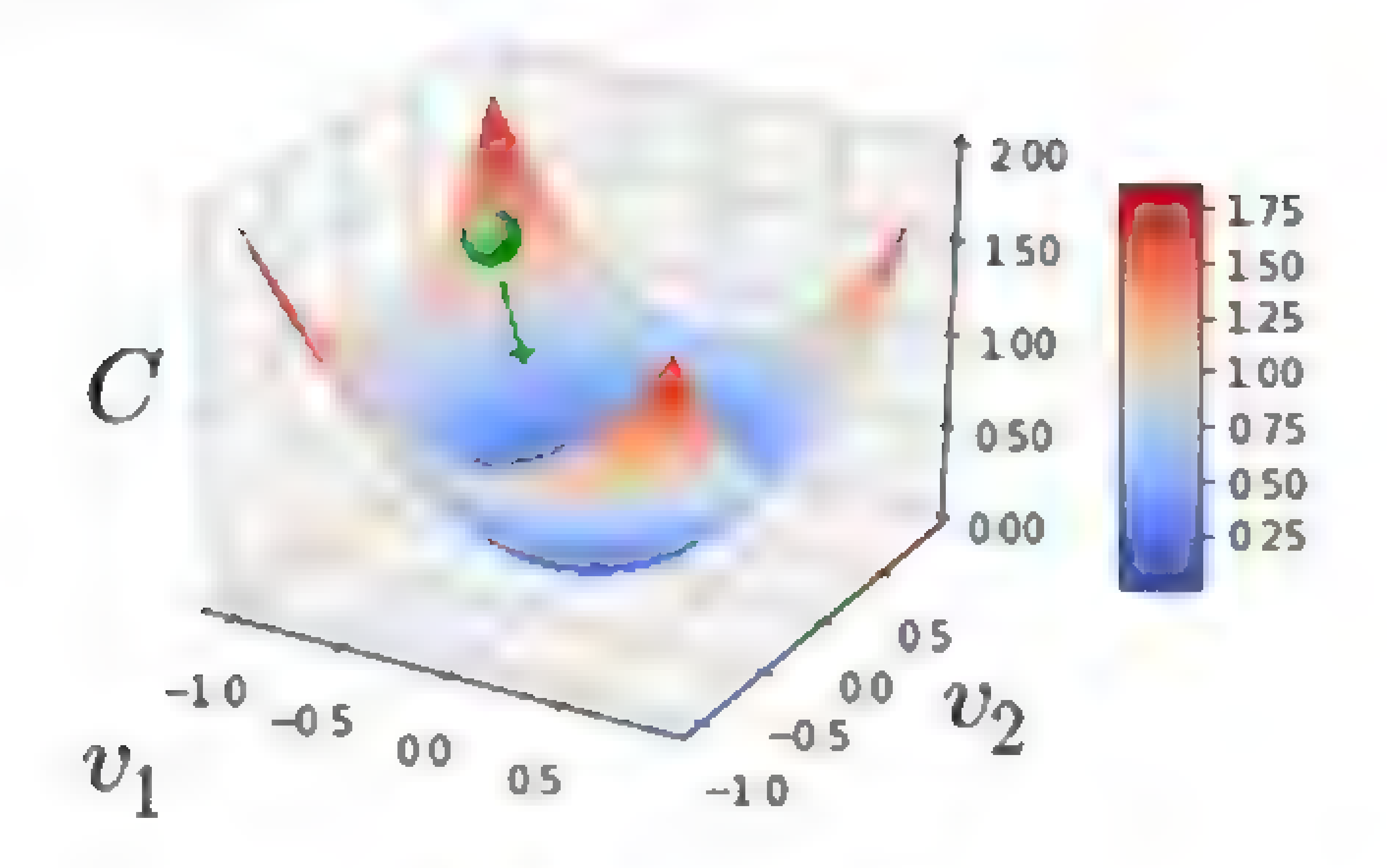


图 2-13 球滚下斜坡示意图

同样，我们应该注意到二次损失函数只是一种方法，其实还有许多其他方法来定义损失。但不管选择哪种方法，最终，我们选择不同损失函数的目的是为了得到：

（1）对权重值的平滑偏导数。

（2）一个良好的凸曲线，可以达到全局最小值。然而，在寻找全局最小值时，还有许多其他因素也在发挥作用。

我们随机选择一个（假想的）球的起点，然后模拟球在向下滚动到山谷底部时的运动。与之类比的情景中，假设我们初始化网络的权重值，或者一般来说，在曲线上的某个任意点上初始化函数的参数（就像在斜坡的任何一点上放一个球），然后检查附近的斜率（导数）。

我们知道，由于重力的作用，球会朝着最大坡度的方向下落。同样，在该点沿导数方向移动权重值，并根据以下规则更新权重值：

设 $J(w)$ = 成本作为权重值的函数，w =网络参数（$v1$ 和 $v2$），w_i=初始权重值集（随机初始化）。

$$w_{update} = w_i - \eta dJ(w)/dw$$

这里，$dJ(w)/dw$ 为权重值 w 对 $J(w)$的偏导数，η 是学习率（Learning Rate）。

学习率更多的是一个超参数，这里虽然没有固定的方法来找到最合适的学习率，但是我们总是可以通过批量损失来找到它。

一种方法是看到损失并分析损失的模式。一般来说，较差的学习率会导致小批量的不稳定损失。它（损失）可以递归地上升和下降，而不需要稳定。

图 2-14 给出了一个更直观的解释。

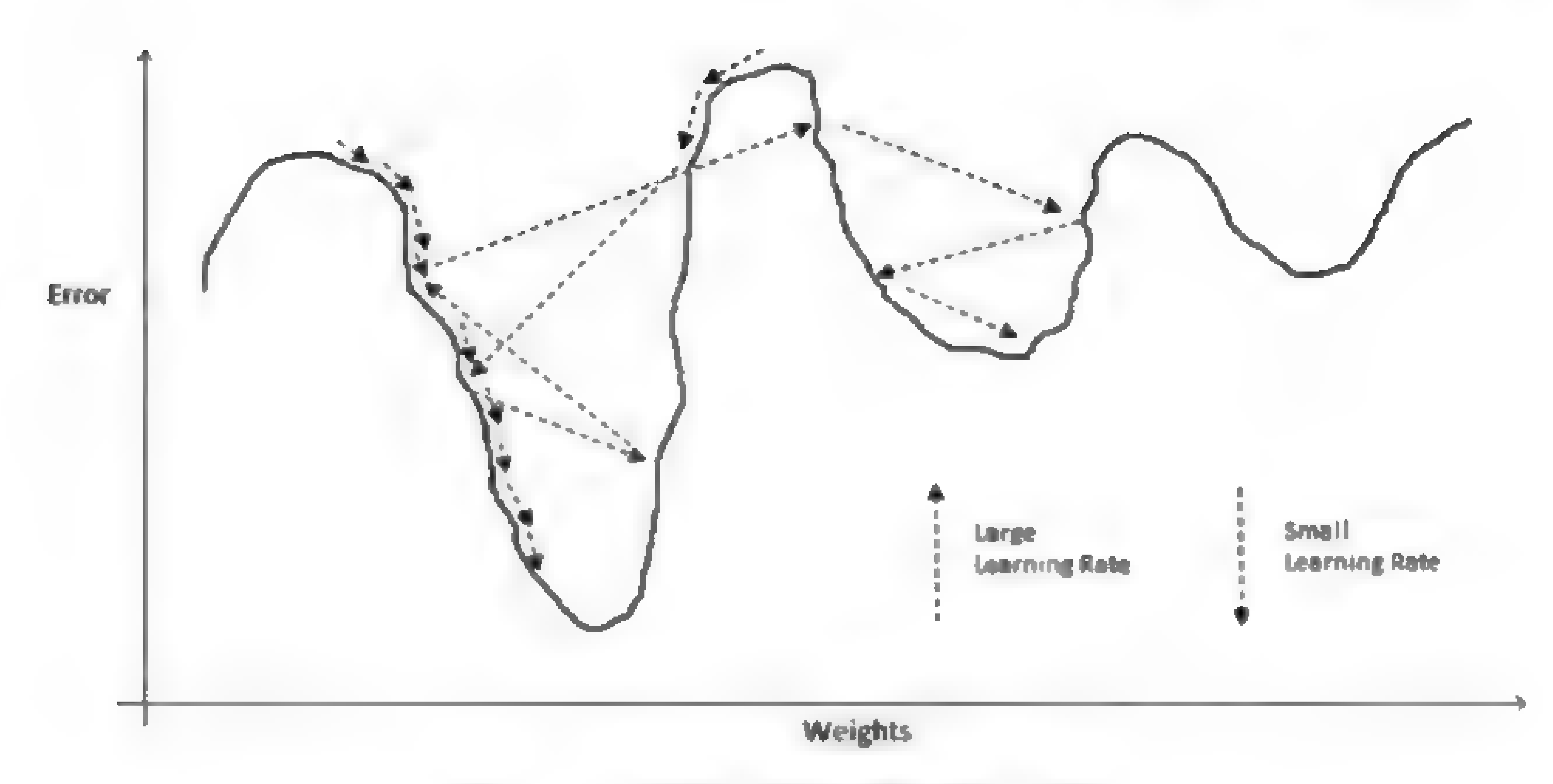


图 2-14　小型和大型学习率影响的示意图

在图 2-14 中，有两种情况：情景 1，小的学习速率；情景 2，大的学习速率。我们的目标是为了达到图 2-14 中的最小值，所以必须达到谷底（就像图 2-13 中球的情况那样）。学习率则与球滚下山时的跳跃幅度有关。

首先考虑情景 1（图 2-14 中的左侧部分），我们在其中进行小的跳跃，逐渐继续向下滚动，慢慢地，最终达到最小值，而球有可能卡在路上的一些小缝隙中，并且由于无法进行大跳跃而无法逃脱它。

在情景 2（图的右侧部分）中，与曲线的斜率相比，学习速率更大。这是一个次优的策略，实际上可能将我们从山谷中驱逐出去，在某些情况下，这可能是一个良好的开端，可以摆脱局部极小的范围，但如果我们跳过全局最小值，就会让人对其感到失望。

在图 2-14 中，我们实现了局部最小值，但这只是一个例子。这意味着权重值会停留在局部最小值上而错过全局最小值。梯度下降或随机梯度下降不保证能够收敛到神经网络的全局最小值（假设隐藏单元不是线性的），因为损失函数是非凸性的。理想情况是阶跃（步长）继续变化并且拥有更具适应性的特点，在起初时可以略高，然后在一段时间内逐渐减小，直到收敛为止。

2.9　反向传播

理解反向传播（Backpropagation）算法可能需要一些时间，读者也可以跳过本节内容，因为很多软件库都具有自动区分和执行整个训练过程的能力。但是，理解这个算法肯定会让你深入了解与深度学习相关的问题（学习问题、缓慢学习、梯度爆炸、梯度下降）。

让我们首先看一张典型的神经网络结构图，如图 2-15 所示。

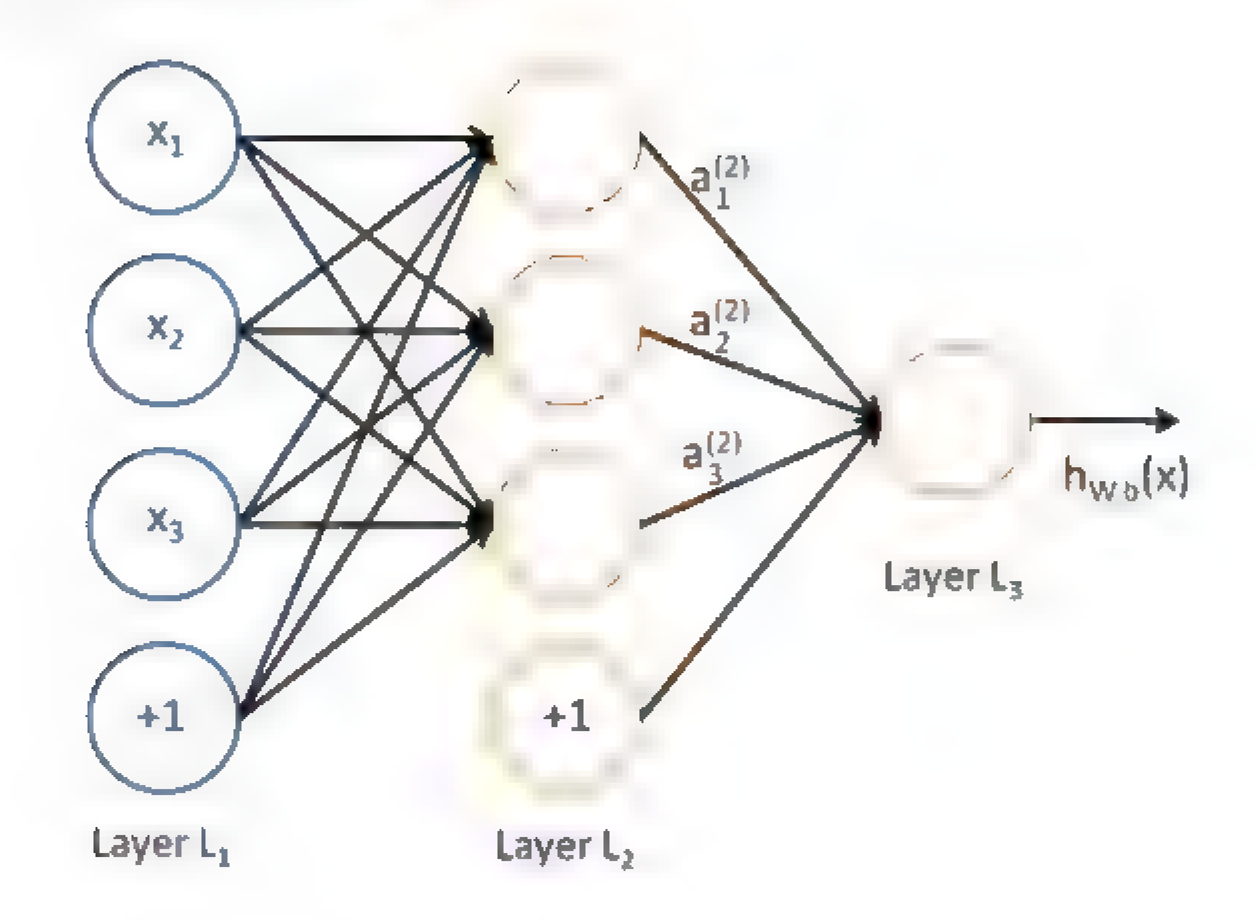


图 2-15　神经网络结构图

图 2-15 是一个包含了输入层 L1、隐藏层 L2 和输出层 L3 的简单神经网络，它的处理流程为根据输入层的 input 以及相应的权重值和配置（图中黑色带箭头的边），通过隐藏层的加工，最终将结果映射到输出层得到结果的输出。模型可以抽象表示为$y = f(x)$，x, y分别表示输入和输出向量。

根据神经网络的处理流程，我们如果要得到输出y，就必须知道图 2-15 中每条边的参数值，这也是神经网络中最重要的部分。在神经网络中是通过迭代的方法来计算这些参数的，具体来讲就是，首先初始化这些参数，通过神经网络的前向传导过程来计算得到输出y，但这些值与真实值存在着误差，我们假设累计误差函数为$err(x)$，然后利用梯度方法极小化$err(x)$来更新参数，直至误差值达到符合要求而停止计算。在更新参数这一过程中，我们就用到了著名的反向传播算法。

为了更好地去说明神经网络的传播算法，我们这里取图 2-15 中的一条路径来做说明，但这不失一般性，假设路径如图 2-16 所示。

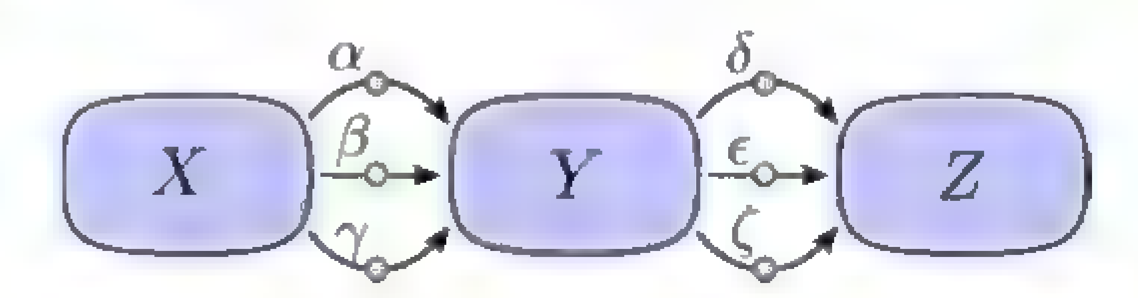


图 2-16　神经网络传播路径示例

图 2-16 中的边表示偏导数，如$\alpha = (\partial Y/\partial X)$。

想要知道输入 X 对输出 Z 的影响，我们可以用偏导数$(\partial Z/\partial X) = (\partial Z/\partial Y) \cdot (\partial Y/\partial X)$，即：

$$(\partial Z/\partial X) = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta \qquad (2.5)$$

我们如果直接使用链式法则进行求导就会遇到一个问题，当路径数量增加时，式（2.5）中的子项目数会呈指数增长，所以这时我们需要把上式右侧部分进行合并，合并后，我们只需要进行一次乘法就可以获得所需要的结果，这样大幅度提升了模型的运算效率，合并后的式子如下：

$$(\partial Z/\partial X) = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta) \qquad (2.6)$$

接下来，我们解决式（2.6）的实现问题。根据计算方向的不同，可以分为正向微分与反向微

分。我们先看针对图 2-16 的正向微分算法，如图 2-17 所示。

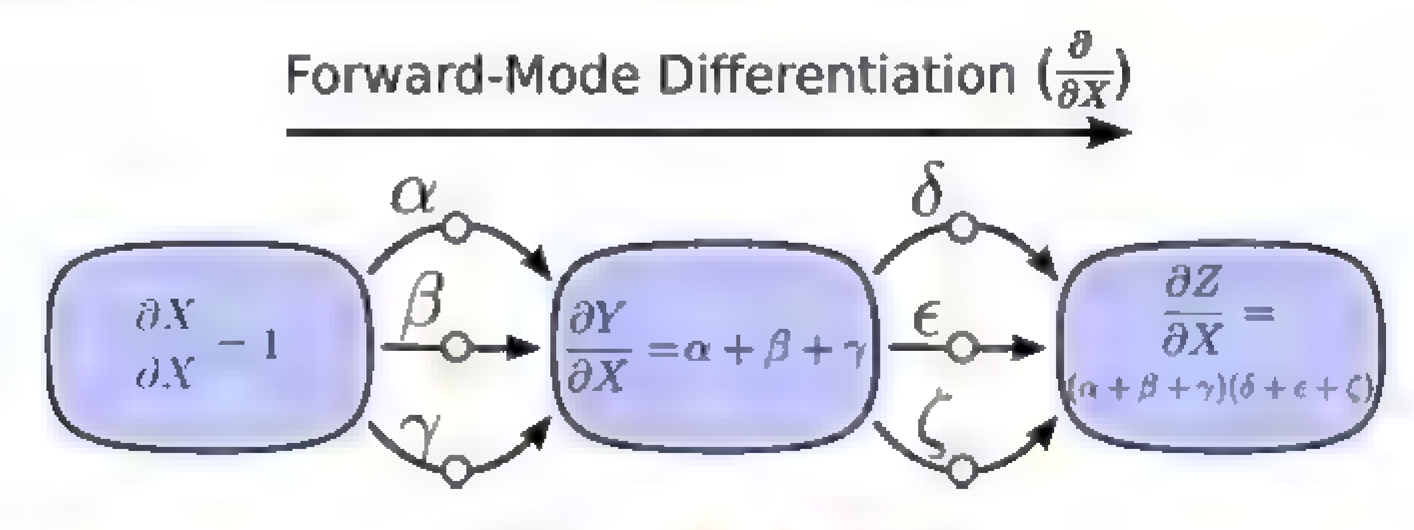


图 2-17　正向微分算法

可以看到，正向微分算法根据路径的传播方向，依次计算路径中的各结点对输入 X 的偏导数，结果中保留了输入对各结点的影响。

下面，我们看一下反向微分算法（见图 2-18）。

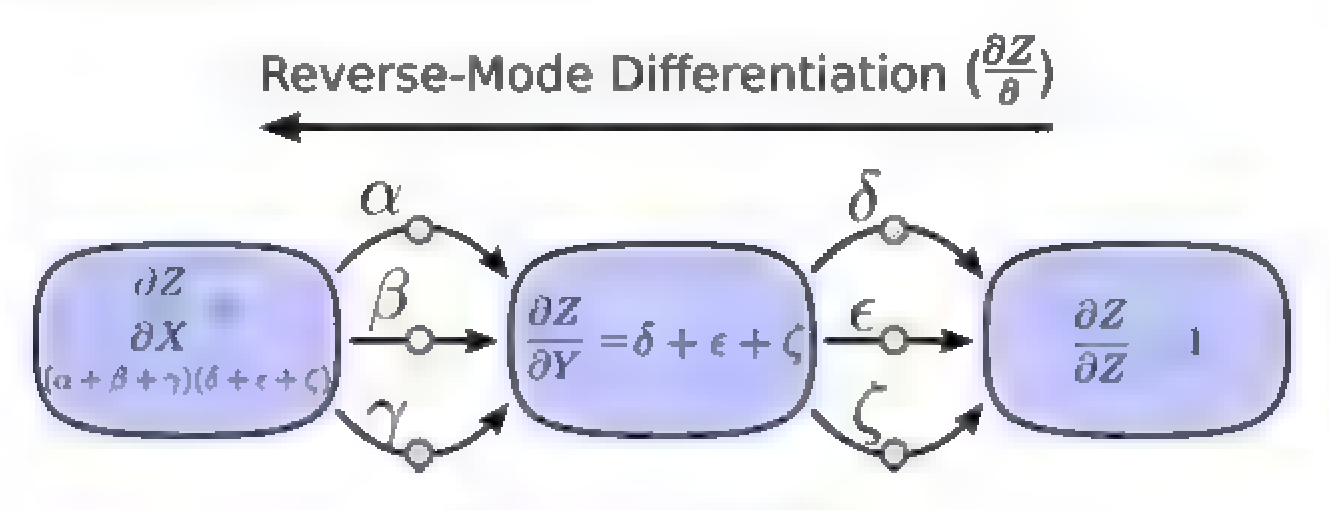


图 2-18　反向微分算法

可以看到，该算法从后向前进行计算，结果中保留了路径中各结点对输出的影响。

这里就有一个问题了，既然正向反向都可以实现式（2.6）的计算，那么我们应该选择哪个算法来实现呢？答案是反向微分算法，理由如下：

首先我们看一个计算式子$e = (a + b) * (b + 1)$的图模型（见图 2-19）。

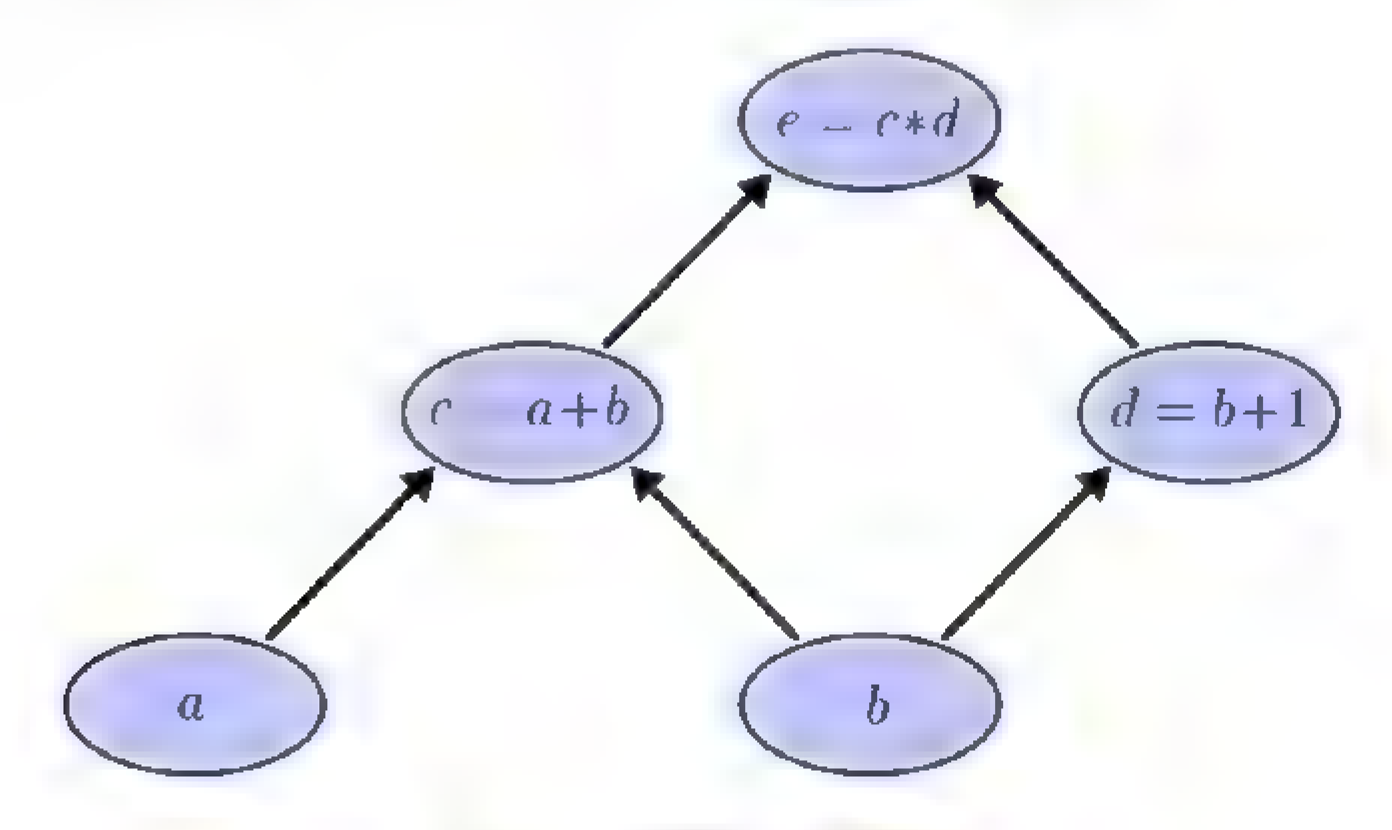


图 2-19　图模型计算示例

其中，c,d 表示中间结果，边的方向表示一个结点是另一个结点的输入。

假设输入变量 a=2、b=1，图 2-19 中各结点的偏导计算结果如图 2-20 所示。

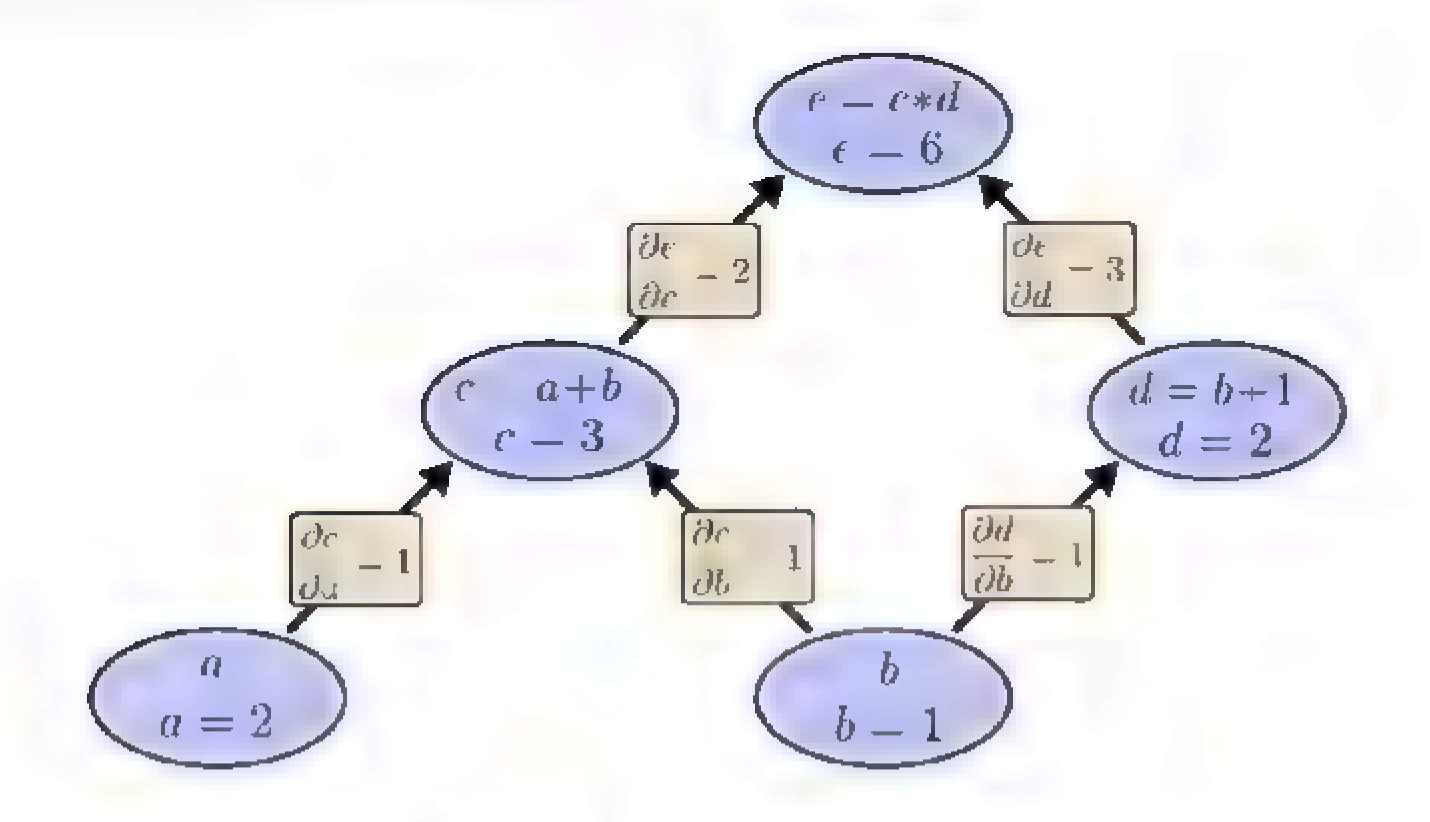


图 2-20 利用正向微分算法得到各结点的偏导计算结果

利用正向微分算法，我们得到关于变量 b 的偏导计算结果如图 2-21 所示。

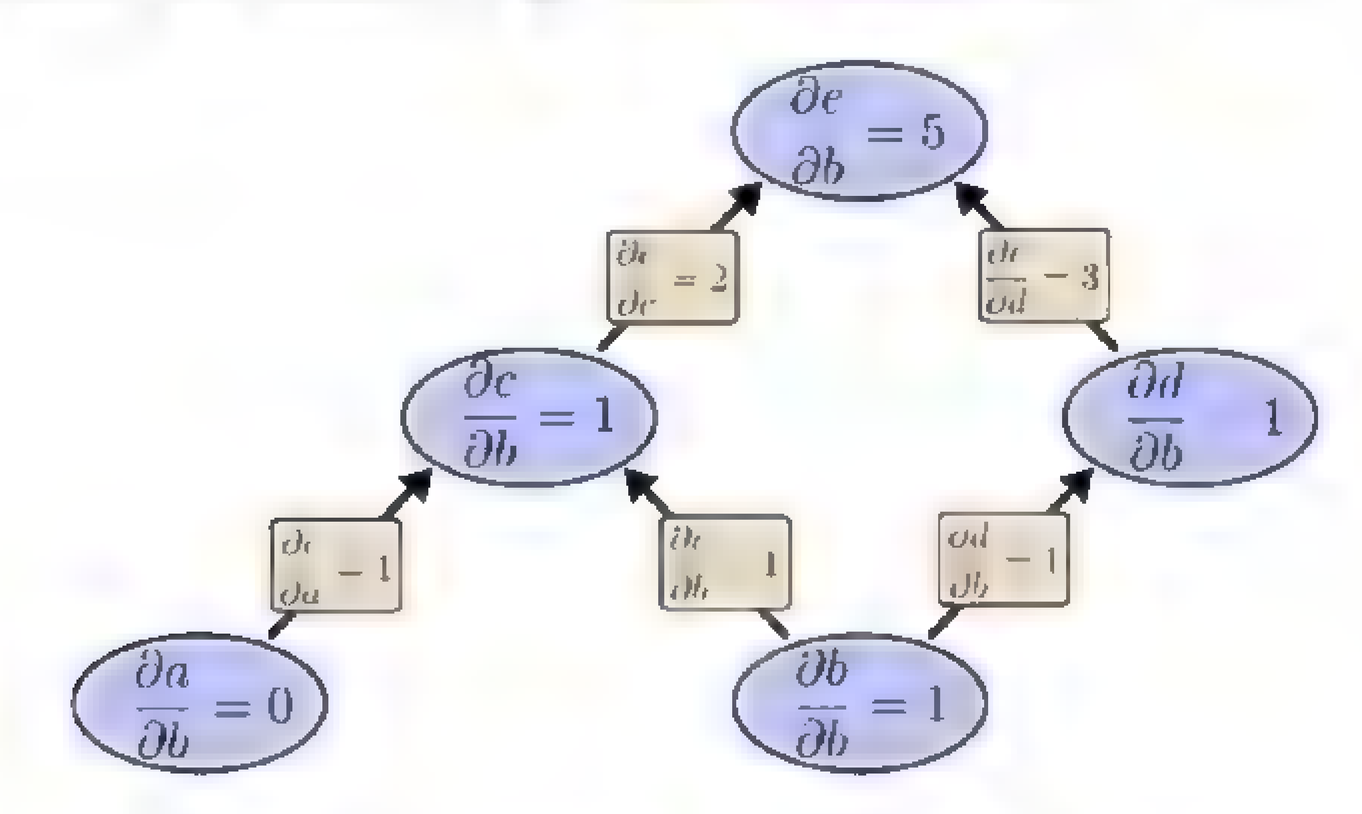


图 2-21 利用正向微分算法得到变量 b 的偏导计算结果

利用反向微分算法，我们得到的偏导计算结果如图 2-22 所示。

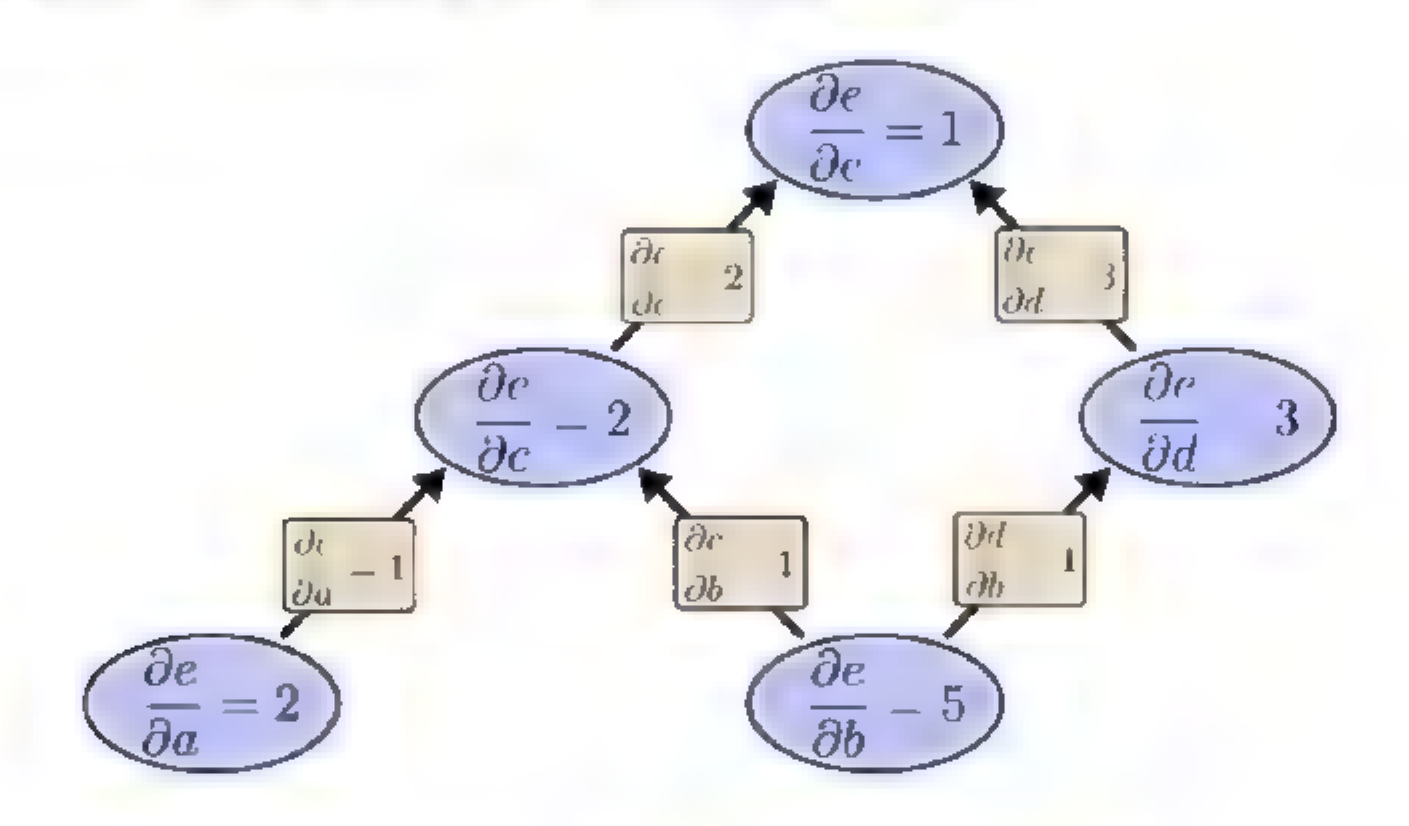


图 2-22 利用反向微分算法得到的偏导计算结果

由此可见，反向微分算法保留了所有变量（包括中间变量）对结果 e 的影响。若 e 为误差函数，则对图进行一次计算，可以得出所有结点对 e 的影响，也就是梯度值，下一步就可以利用这些梯度

值来更新边的权重值了；而正向微分算法得到的结果是只保留了一个输入变量对误差 e 的影响，显然，想要获得多个变量对 e 的影响，我们就需要进行多次计算。所以正向微分算法在效率上明显不如反向微分，这也是我们选择反向微分算法的原因。

2.10　总结

在本章，我们首先对深度学习的理念和产生的历史背景做了说明，认识到深度学习在人工智能当中起到非常重要的作用。

其次，我们为了对深度学习的内部原理进行解读，介绍了其中的关键部分——神经元模型。通过神经元模型的解读，我们对深度学习的基本结构有了认知。

接着，我们从单层神经网络、多层神经网络和 Encoder-Decoder 网络等方面对于深度学习的内部网络结构进行了更深层次的解析，对其内部逻辑和机制运行情况有了大致清晰的认知。因为现实中的问题一般都非常复杂，怎样选择一个高效的优化算法便成为我们着重关注的方法，所以我们就介绍了随机梯度下降算法。

最后，我们介绍了著名的反向传播算法（BP），从数学的角度做简要解读，使我们认识到反向传播算法的魅力所在，为我们进一步学习深度学习或者说神经网络模型奠定了良好的基础。

下一章，我们将介绍本书所使用的技术框架：TensorFlow。

第 3 章

TensorFlow

在本章中，首先我们会对 TensorFlow 的概念、主要特征、安装及其三个组成部分进行详细解读，梳理出这些核心概念之间的关系。其次，在了解了一些核心概念之后，我们会对 TensorFlow 的工作原理进行深度解析，将 TensorFlow 平台的“client—master—worker”架构底层中涉及到的内在逻辑和技术路径进行详细解读，以便我们对 TensorFlow 框架的运行机制有更多的认知。接着，我们会利用示例对 TensorFlow 客户端进行专门解读，读者会在示例中感受 TensorFlow 客户端内在的运行情况。然后，我们会对 TensorFlow 中的常见元素逐一详解，并做一些对比分析。最后，我们对 TensorFlow 的变量作用域机制做一些介绍，并实现一个三层神经网络对 MNIST 数字数据集进行分类的例子。

本章中涉及的完整代码，请查看文件夹 ch3 下的代码文件：3_tensorflow_introduction.ipynb。

3.1 TensorFlow 概念解读

由于深度学习研究的最终目的是为了解决我们现实世界中遇到的困难，因此在学术界不断探索新理念、新模型的同时，业界优秀的从业人员也在加速发布各类可实现的技术框架，这样深度学习框架就在学术界和工业界的相互推动下得以迅速发展起来。这里给出谷歌公司经过长期的研究和尝试，在实践的基础上推出的目前最优秀的深度学习框架之一——TensorFlow。TensorFlow 最初是由谷歌公司的大脑小组（隶属于谷歌机器智能研究机构）的研究员和工程师们开发出来的，是一个采用数据流图（Data Flow Graph）的开源分布式数值计算框架，主要用于减轻实现神经网络细节层面的工作量（例如，计算神经网络权重值的导数）。TensorFlow 通过使用统一计算设备架构（Compute Unified Device Architecture，CUDA）对数值进行有效计算，该架构是由 NVIDIA 公司引入的并行计算平台。

TensorFlow 主要是由计算图、张量以及模型会话三个部分组成的。TensorFlow 中的计算可以表示为一个计算图（Computation Graph），或者称作有向图（Directed Graph），图中的每一个数学运算或操作（Operation）都将被视为一个节点（Node），而节点与节点之间的连接被称为边（Edge），

图中的这些边（Edge）则表示在节点间相互关联的流动（Flow）的多维数据数组，即张量（Tensor）。这个计算图表述了数据的计算流程，它负责维护和更新状态，我们可以对计算图中的分支进行条件控制或循环操作。简单说，我们可以用张量表示数据，用计算图搭建神经网络，用会话执行计算图，在优化线上的权重值（参数）后得到我们的模型。

这里，我们还可以使用多种语言对计算图进行设计。TensorFlow 这样灵活的架构让我们可以在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等。作为优秀的技术架构，TensorFlow 在机器学习方面解决了以下几个核心问题：

（1）使用 TensorFlow 工具可以利用很简洁的语言来实现各类复杂算法模型，这样一来，我们就可以从日常中极为消耗精力的编码、调试工作中解放出来。

（2）由于 TensorFlow 内核执行系统使用的是 C++，因此在执行效率方面非常高效。

（3）TensorFlow 极佳的分层架构使得模型能够很方便地运行在异构设备之上。

（4）TensorFlow 包含了 TensorBoard 等极好的配套工具，且第三方社区也在不断贡献很多实用的辅助工具，例如 TFLearn 等，这些工具使得代码变得更加简洁、数据处理工作更便捷。

如果读者感兴趣，可以参考下面的网址以查阅关于 TensorFlow 方面的内容。

- TensorFlow 官网： tensorflow.google.cn。
- TensorFlow 的应用程序编程接口（API）： tensorflow.google.cn/api_docs/python。
- GitHub 网址：github.com/tensorflow。
- 模型仓库：github.com/tensorflow/models。

3.2　TensorFlow 主要特征

3.2.1　自动求微分

基于梯度的机器学习算法会受益于 TensorFlow 自动求微分的能力。在使用 TensorFlow 时，我们只需要定义预测模型的结构，将这个结构和目标函数（Objective Function）结合在一起，并添加数据，TensorFlow 将自动为我们计算相关的微分导数。计算某个变量相对于其他变量的导数仅仅是通过扩展你的计算图来完成的，所以我们可以一直很清晰地看到系统中究竟在发生什么。

3.2.2　多语言支持

TensorFlow 有一个合理的 C++使用界面，也有一个易用的 Python 使用界面来构建和执行我们的计算图（Computation Graph）。我们可以直接编写 Python/C++程序，也可以用交互式的 iPython 界面来调用 TensorFlow 以尝试一些想法，它可以帮我们将笔记、代码、可视化等有条理地归置好。当然这仅仅是一个起点，它同时也在促使我们创造自己最喜欢的语言界面，比如 Go、Java、Lua、JavaScript、R 等。

3.2.3 高度的灵活性

TensorFlow 不是一个严格意义上的“神经网络”库。如果我们能够将计算表示为一个数据流图，那么就可以使用 TensorFlow 来构建图和描写驱动计算的内部循环。TensorFlow 提供了有用的工具来帮助你组装“子图”（常用于神经网络），当然用户也可以自己在 TensorFlow 基础上编写自己的“上层库”。定义顺手好用的新复合操作和编写一个 Python 函数一样容易，而且也不用担心性能损耗。当然万一我们发现找不到想要的底层数据操作，我们也可以自己编写一点 C++代码来丰富底层的操作。

3.2.4 真正的可移植性

TensorFlow 在 CPU 和 GPU 上运行，比如说可以运行在台式机、服务器、移动设备等上面。在没有特殊硬件的前提下，TensorFlow 能够帮我们在自己的笔记本上运行一下机器学习、深度学习模型。TensorFlow 也可以帮助我们将自己的训练模型在多个 CPU 上进行规模化的运算，同时不必修改代码。TensorFlow 还可以帮助我们将自己训练好的模型作为产品的一部分部署到手机 App 里。TensorFlow 更可以帮助我们将自己的模型作为云端服务运行在自己的服务器上，或者运行在 Docker 容器里。

3.2.5 将科研和产品联系在一起

过去，要将科研中的机器学习想法用到产品中，需要大量的代码重写工作。现在，谷歌公司的科学家用 TensorFlow 尝试新的算法，产品团队则用 TensorFlow 来训练和使用计算模型，并直接提供给在线用户。使用 TensorFlow 可以让应用型研究者将想法迅速地运用到产品中，也可以让学术性研究者更直接地彼此分享代码，从而提高科研产出率。

3.2.6 性能最优化

例如，现在你有一个带有 32 个 CPU 内核、4 个 GPU 显卡的工作站，就可以利用 TensorFlow 将工作站的计算潜能全部发挥出来。TensorFlow 给予了线程、队列、异步操作等最佳的支持，能够让我们将自己手边硬件的计算潜能全部发挥出来。我们可以自由地将 TensorFlow 图中的计算元素分配到不同设备上，让 TensorFlow 帮我们管理好这些不同的副本。

3.3 TensorFlow 安装

我们可以在多个平台上安装和使用 TensorFlow，例如 Linux、Mac OS 和 Windows。当然，我们也可以使用 TensorFlow 最新的 GitHub 源来构建和安装 TensorFlow。此外，如果我们的电脑运行

的是 Windows 系统，那么比较常用的是通过 Anaconda 来安装 TensorFlow。

由于 Python 3 附带了 pip3 包管理器，它是用于安装 TensorFlow 的程序，因此如果我们使用的是 Python 的这个版本，就无须安装 pip。为简单起见，在本节中将展示如何使用本机 pip 安装 TensorFlow。下面给出在 Windows 系统下安装 TensorFlow 的两个命令。启动一个“终端”程序，然后在该“终端”中执行相应的 pip3 install 命令。

（1）要安装 TensorFlow 的 CPU 版本，输入以下命令：

```
C:\> pip3 install --upgrade tensorflow
```

（2）要安装 TensorFlow 的 GPU 版本，输入以下命令：

```
C:\> pip3 install --upgrade tensorflow-gpu
```

在 Linux 方面，TensorFlow Python API 支持 Python 2.7 和 Python 3.3+，因此我们需要安装 Python 才能启动 TensorFlow 安装。我们必须安装 Cuda Toolkit 7.5 和 cuDNN v5.1 +才能获得 GPU 支持。笔者使用的是 Ubuntu 系统，是采用 pip3 的命令来安装 TensorFlow 的，更多信息也可以参考 https://tensorflow.google.cn/install/source 上的说明。

在 Linux 上安装 TensorFlow

这里将给出如何在 Ubuntu 14.04 或更高版本上安装 TensorFlow。此处提供的说明可能也适用于 Linux 其他版本，只需进行少量的调整即可。

但是，在继续执行正式步骤之前，我们需要确定在平台上安装哪个 TensorFlow，我们可以在 GPU 和 CPU 上运行数据密集型张量应用程序。因此，应该选择以下 TensorFlow 两种版本中的一个，在我们的平台上进行安装：

- 仅支持 CPU 的 TensorFlow：如果计算机上没有安装 NVIDIA 等 GPU，就必须使用此版本安装并开始计算。这个很简单，可以在 5 到 10 分钟内完成。
- 支持 GPU 的 TensorFlow：深度学习应用程序通常需要非常高的计算资源，TensorFlow 也不例外，我们通常可以在 GPU 上而不是在 CPU 上加快数据计算和分析速度。如果你的计算机上有 NVIDIAGPU 硬件，你应该安装并使用此版本。

根据我们的经验，即使计算机上集成了 NVIDIA GPU 硬件，也有必要安装并首先尝试仅使用 CPU 的版本，如果你没有体验到良好的性能，那么再切换到 GPU 来进行支持。

支持 GPU 的 TensorFlow 版本有几个要求，例如 64 位 Linux、Python 2.7（或 Python 3 的 3.3+），NVIDIACUDA7.5 或更高版本（Pascal GPU 需要 CUDA 8.0）和 NVIDIA cuDNN v4.0（最小）或 v5.1（推荐）。更具体地说，TensorFlow 的当前开发仅支持使用 NVIDIA 工具包和软件的 GPU 计算。因此，必须在 Linux 计算机上安装以下软件才能使得预测分析应用程序获得 GPU 的支持：

- Python
- NVIDIA Driver

- CUDA with compute capability >= 3.0
- CuDNN
- TensorFlow

1. 安装 Python 和 NVIDIA 驱动程序

在不同的平台上安装 Python 的操作相对比较简单，这里不再赘述。另外，假设你的计算机中已经安装了 NVIDIA GPU。

要检查一下你的计算机中的 GPU 是否已正确安装和正常工作，请在终端上执行以下命令：

```
$ lspci -nnk | grep -i nvidia
```

由于预测分析很大程度上依赖于机器学习和深度学习算法，因此请确保你的计算机上安装了一些基本软件包，例如 GCC 和一些用于科学计算的 Python 软件包。

只需在终端上执行如下命令：

```
$ sudo apt-get update
$ sudo apt-get install libglu1-mesa libxi-dev libxmu-dev -y
$ sudo apt-get — yes install build-essential
$ sudo apt-get install python-pip python-dev -y
$ sudo apt-get install python-numpy python-scipy -y
```

现在通过 wget 下载 NVIDIA 驱动程序（不要忘记为你的计算机选择正确的版本），并以 silent 模式运行脚本：

```
$ wget
http://us.download.nvidia.com/XFree86/Linux-x86_64/367.44/NVIDIA-Linux-x86_64-367.44.run
$ sudo chmod +x NVIDIA-Linux-x86_64-367.35.run
$ ./NVIDIA-Linux-x86_64-367.35.run --silent
```

一些 GPU 显卡，如 NVidia GTX 1080，附带内置驱动程序。因此，如果你的计算机具有不同于 GTX 1080 的 GPU，就必须下载该 GPU 的驱动程序。

要确保驱动程序是否已正确安装，请在终端上执行以下命令：

```
$ nvidia-smi
```

命令的结果应如图 3-1 所示。

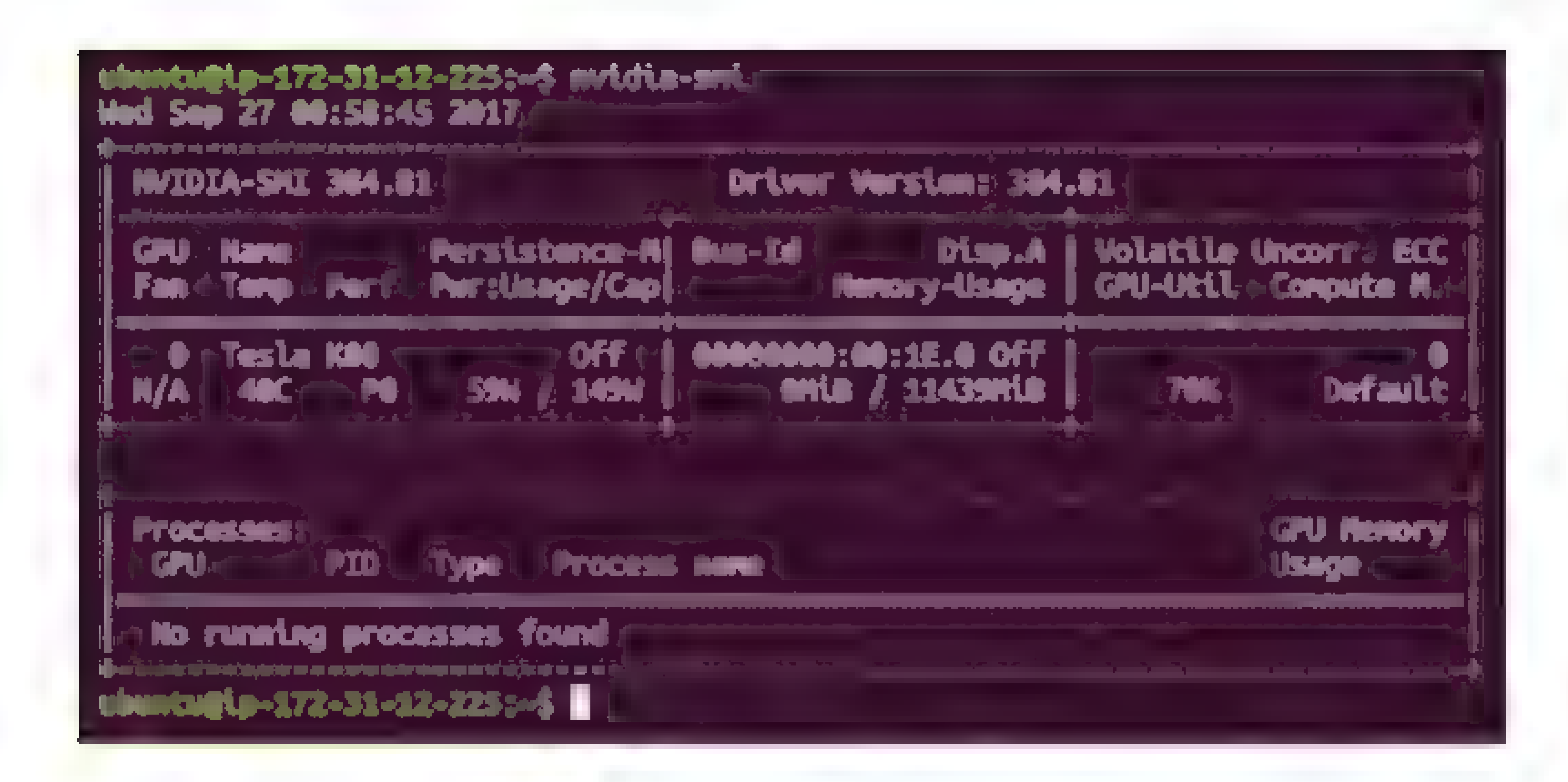


图 3-1 nvidia-smi 命令的输出结果

2. 安装 NVIDIA CUDA

为了将 TensorFlow 与 NVIDIA GPU 配合使用，需要安装 CUDA Toolkit 8.0 及其相关的 NVIDIA 驱动程序。CUDA 工具包包括：

- GPU 加速库，如用于快速傅里叶变换的 cuFFT（FFT）。
- 基本线性代数子程序（BLAS）的 cuBLAS。
- 稀疏矩阵例程的 cuSPARSE。
- 用于密集和稀疏的直接求解器的 cuSOLVER。
- 用于随机数生成的 cuRAND，用于图像的 NPP 和视频处理原语。
- NVIDIA 图形分析库的 nvGRICH。
- 推力模板并行算法和数据结构专用的 CUDA 数学库。

对于 Linux，下载和安装所需的包：

https://developer.nvidia.com/cuda-downloads

使用的 wget 命令如下：

```
$ wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_
installers/cuda_8.0.61_375.26_linux-run
$ sudo chmod +x cuda_8.0.61_375.26_linux.run
$ ./ cuda_8.0.61_375.26_linux.run --driver --silent
$ ./ cuda_8.0.61_375.26_linux.run --toolkit --silent
$ ./ cuda_8.0.61_375.26_linux.run --samples -silent
```

另外，确保你已经将 CUDA 安装路径添加到 LD_LIBRARY_PATH 环境变量中：

```
$ echo 'export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/
usr/local/cuda/extras/CUPTI/lib64"' >> ~/.bashrc
$ echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
$ source ~/.bashrc
```

3. 安装 NVIDIA cuDNN v5.1+

安装CUDA Toolkit后，从Linux上下载cuDNN v5.1库，解压缩文件并将其复制到CUDA Toolkit目录（假设在/ usr / local / cuda /中）：

```
$ cd /usr/local
$sudo mkdir cuda
$ cd ~/Downloads/
$ wget http://developer2.download.nvidia.com/compute/machine-learning/
cudnn/secure/v6/prod/8.0_20170427/cudnn-8.0-linux-x64-v6.0.tgz
$ sudo tar -xvzf cudnn-8.0-linux-x64-v6.0.tgz
$ cp cuda/lib64/* /usr/local/cuda/lib64/
$ cp cuda/include/cudnn.h /usr/local/cuda/include/
```

注意，要安装 cuDNN v5.1 库，我们必须在 https://developer.nvidia.com/accelerated-computing-developer 上注册加速计算开发程序。现在，当安装了 cuDNN v5.1 库之后，请确保创建了 CUDA_HOME 环境变量。

4. 安装 libcupti-dev 库

最后，需要在你的计算机上安装 libcupti-dev 库，这是提供高级分析支持的 NVIDIA CUDA。要安装此库，请执行以下命令：

```
$ sudo apt-get install libcupti-dev
```

5. 安装 TensorFlow

有关如何为 CPU 安装最新版本的 TensorFlow，以及如何为具有 NVIDIA 、cuDNN 和 CUDA 计算能力的 GPU 提供支持，请参阅下面的部分。可以在计算机上以多种方式安装 TensorFlow，比如使用 virtualenv、pip、Docker 和 Anaconda。这里使用 pip 和 virtualenv 方式。（有兴趣的读者可以尝试使用 Docker 和 Anaconda 安装，详见 https://tensorflow.google.cn/install/。）

（1）使用本地 pip 安装 TensorFlow

对于 Python 2.7，仅支持 CPU 版本的，具体如下：

```
$ pip install tensorflow
# For Python 3.x and of course with only CPU support:
$ pip3 install tensorflow
# For Python 2.7 and of course with GPU support:
$ pip install tensorflow-gpu
# For Python 3.x and of course with GPU support:
$ pip3 install tensorflow-gpu
```

（2）使用 virtualenv 安装 TensorFlow

如果你已经在系统上安装了Python 2+(或3+)和pip(或pip3)，请按照以下步骤安装TensorFlow。

①创建 virtualenv 环境：

```
$ virtualenv - system site packages targetDirectory
```

targetDirectory 表示 virtualenv 树的根目录。默认情况下，它是~/ tensorflow（也可以选择任何其他目录）。

②按如下方式激活 virtualenv 环境：

```
$ source ~/tensorflow/bin/activate        # bash, sh, ksh, or zsh
   $ source ~/tensorflow/bin/activate.csh        # csh or tcsh
```

如果命令在步骤 2 中执行成功，那么在“终端”程序中上应该可以看到以下内容：

```
(tensorflow)$
```

③安装 TensorFlow。

按照以下命令之一在激活的 virtualenv 环境中安装 TensorFlow。

对于仅支持 CPU 的 Python 2.7，请使用以下命令：

```
(tensorflow)$ pip install --upgrade tensorflow
```

对于仅支持 CPU 的 Python 3.x，请使用以下命令：

```
(tensorflow)$ pip3 install --upgrade tensorflow
```

对于支持 GPU 的 Python 2.7，请使用以下命令：

```
(tensorflow)$ pip install --upgrade tensorflow-gpu
```

对于仅支持 GPU 的 Python 3.x，请使用以下命令：

```
(tensorflow)$ pip3 install --upgrade tensorflow-gpu
```

如果上述命令成功，就跳过步骤 5；如果上述命令失败，请执行步骤 5。此外，如果步骤 3 以某种方式失败，请尝试通过执行以下命令以在激活的 virtualenv 环境中安装 TensorFlow：

```
#对于 python 2.7（选择具有 CPU 或 GPU 支持的适当 URL）：
(tensorflow)$ pip install --upgrade TF_PYTHON_URL
#对于 python 3 .x（选择具有 CPU 或 GPU 支持的适当 URL）：
(tensorflow)$ pip3 install --upgrade TF_PYTHON_URL
```

④验证安装。

要在步骤 3 中验证安装，必须激活虚拟环境。如果 virtualenv 环境当前未处于激活状态，请执行以下命令之一：

```
$ source ~/tensorflow/bin/activate     # bash, sh, ksh, or zsh
$ source ~/tensorflow/bin/activate.csh    # csh or tcsh
```

⑤卸载 TensorFlow。

要卸载 TensorFlow，只需删除创建的目录树即可。例如：

```
$ rm -r targetDirectory
```

6. 测试安装的 TensorFlow

启动一个 Python 终端（只需在终端上输入 python 或 python3），执行一段常见的“Hello, TensorFlow!”程序，代码行如下：

```
>>> import tensorflow as tf
>>> hello = tf.constant("Hello, TensorFlow!")
>>> sess=tf.Session()
>>> print sess.run(hello)
```

如果 TensorFlow 安装成功，我们将看到输出结果为“Hello, TensorFlow!”。

3.4 TensorFlow 计算图

在执行 TensorFlow 程序时，我们需要构建一个计算图，然后按照计算图启动一个会话，在会话中完成变量赋值、计算等操作并得到最终结果。换句话说，TensorFlow 的计算图可以划分为两部分：

- 构造部分，包含计算流图，用于构建模型。
- 执行部分，通过会话（Session）执行图中的计算，用于提供数据并获得结果。

对于计算图的构造部分而言，我们又可以分为两部分：

- 创建源节点。
- 源节点的输出传递给其他节点做运算操作。

这里，更吸引我们的是，TensorFlow 会在 C ++引擎上执行每一项运算操作，这意味着在 Python 中也不会执行任何乘法或加法，Python 只是一个包装器。从根本上讲，TensorFlow C ++引擎包含以下两方面：

- 卷积、最大池化、Sigmoid 等操作的高效实现。
- 转发模式操作的导数。

计算图基本上类似于数据流图。图 3-2 显示了一个简单计算的计算图，用于计算简单的等式，如 $z = d \times c = (a + b) \times c$。

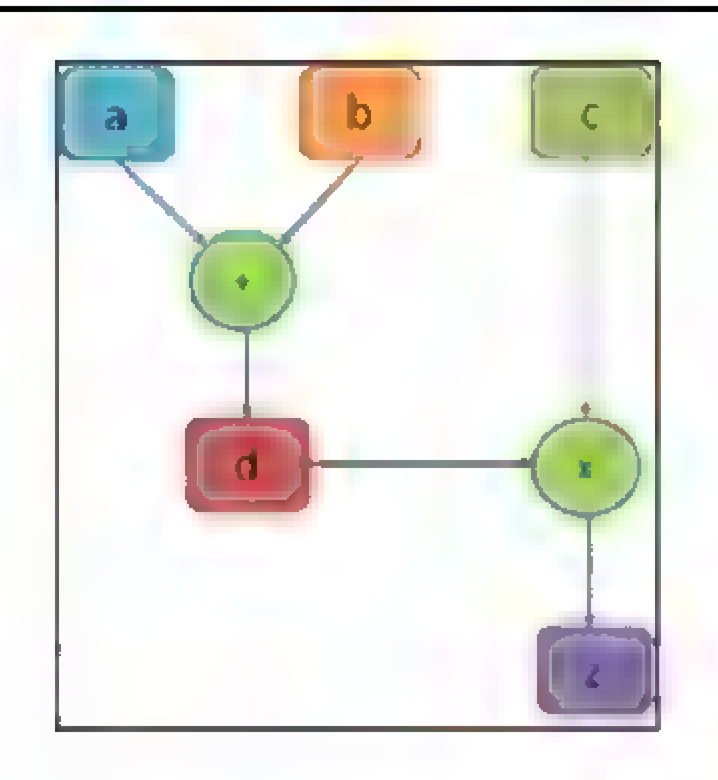


图 3-2　一个简单的执行图

在图 3-2 中，圆圈表示节点处的操作，矩形表示整个数据计算图。在 TensorFlow 的实施中，TensorFlow 计算图包含以下内容：

- 一组 tf.Operation 对象：用于表示要执行的计算单元。
- tf.Tensor 对象：用于表示控制操作之间数据流的数据单元。

使用 TensorFlow 还可以执行延迟操作。一旦在计算图的构建阶段编写了高度组合的表达式，我们仍然可以在会话的运行阶段去对它们求值。从技术上讲，TensorFlow 会给出措施并以有效的方式按时执行。例如，使用 GPU 并行执行代码的独立部分，如图 3-3 所示。

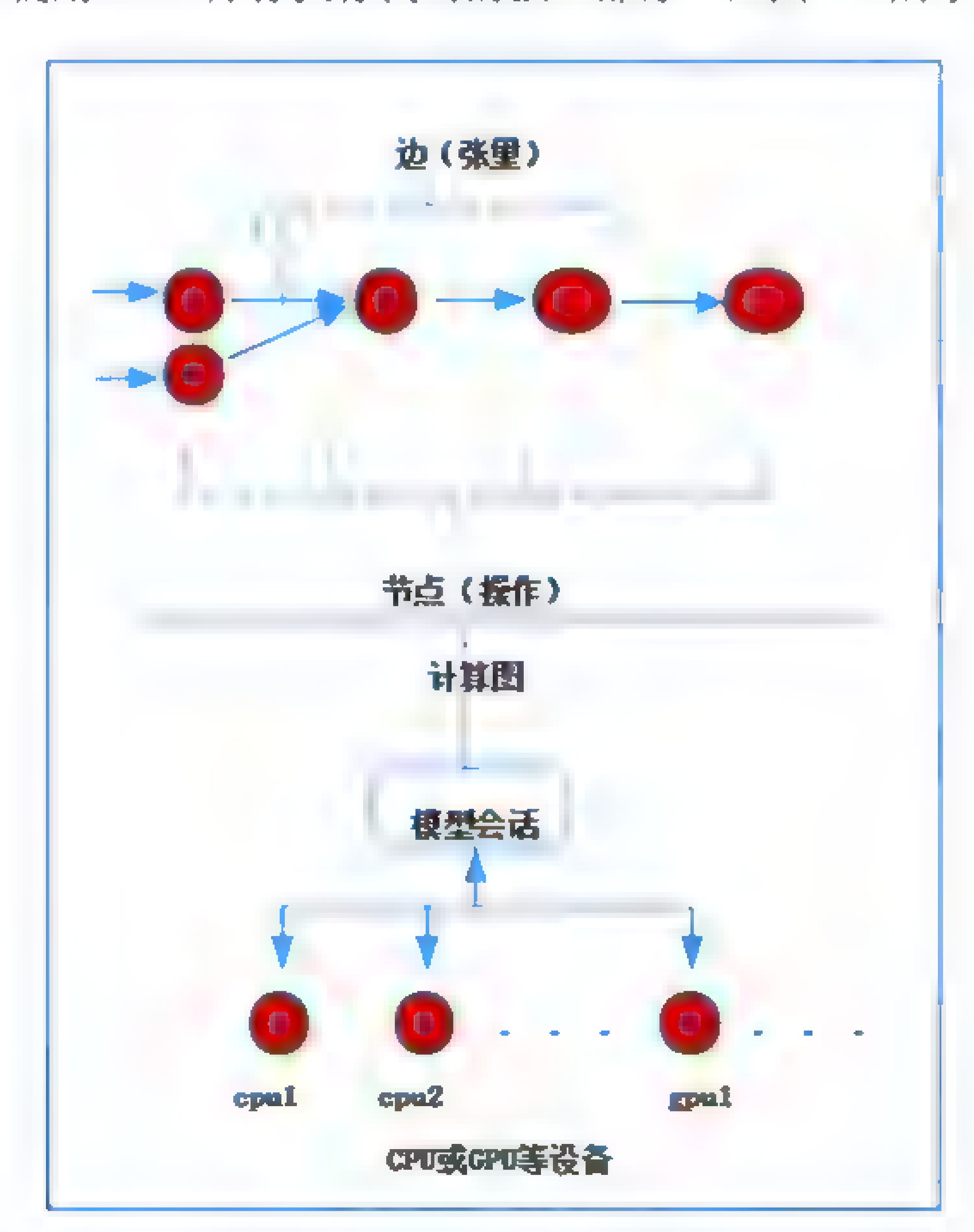


图 3-3　TensorFlow 图中的边和节点将在 CPU 或 GPU 等设备上的会话下执行

3.5 TensorFlow 张量和模型会话

3.5.1 张量

在 TensorFlow 中，张量（Tensor）是对运算结果的引用，运算结果多数情况下是以数组的形式存储的，与 numpy 中的数组不同的是，张量还具有三个重要属性：名字、维度、类型。张量的名字是张量的唯一标识符，我们通过名字可以发现张量是如何被计算出来的。例如“multiply:0”代表的是计算节点“multiply”的第 1 个输出结果，“add:2”代表的是计算节点“add”的第 3 个输出结果。

TensorFlow 有两种类型的边，或者说张量（Tensor）的类型有两种：

- 正常：它们承载节点之间的数据结构。来自一个节点的一个运算或操作的输出变为另一个运算或操作的输入。连接两个节点的线携带这些数值。
- 特殊：此边不携带数值，但是只表示两个节点（比如 X 和 Y）之间的控制依赖关系。这意味着节点 Y 只有在 X 中的运算或操作已经执行时才会被执行。

我们也可以这样理解，TensorFlow 计算图中的每个节点的输入输出都是张量（Tensor），而连接节点的边（有向线段）就是 Flow，表示从一个张量（Tensor）状态到另一个张量（Tensor）状态。我们在编写程序时，都是逐步计算的，每计算完一步便能够获得一个执行结果。对于 Tensor 数据类型而言，虽然我们可以事先定义或者根据计算图的结构推断得到，但是有一类特殊的边中并没有数据流动，这种边便是依赖控制（Control Dependencies），其作用便是让它的起始节点执行完之后再去执行目标节点，这样一来我们就可以使用边进行灵活的条件控制了。比如，我们在 TensorFlow 实施中定义了依赖控制关系，便可以在其他独立的操作之间强制执行排序，以作为限制使用内存最高峰值的一种方式。

在 TensorFlow 中，一个张量基本上是一个 n 维数组，下面让我们通过表 3-1 中的内容来解读一下张量和维数的关系。

表 3-1 张量和维数的关系

维数	阶	名称	示例
0-D	0	标量(Scalar)	s=1 2 3
1-D	1	向量(Vector)	v=[1,2,3]
2-D	2	矩阵	m=[[1,2,3],[4,5,6]]
3-D	3	张量	t=[[...
...	...	...	...
n-D	n	张量	T=[[...

张量可以表示 0 阶到 n 阶的数组（列表）。这里，阶是张量的维数。第 0 阶张量是一个数，也

就是标量；第 1 阶张量是一个一维数组，也就是向量；第 2 阶张量是一个二维数组，也就是矩阵；第 *n* 阶张量是一个 *n* 维数组。

3.5.2 会话

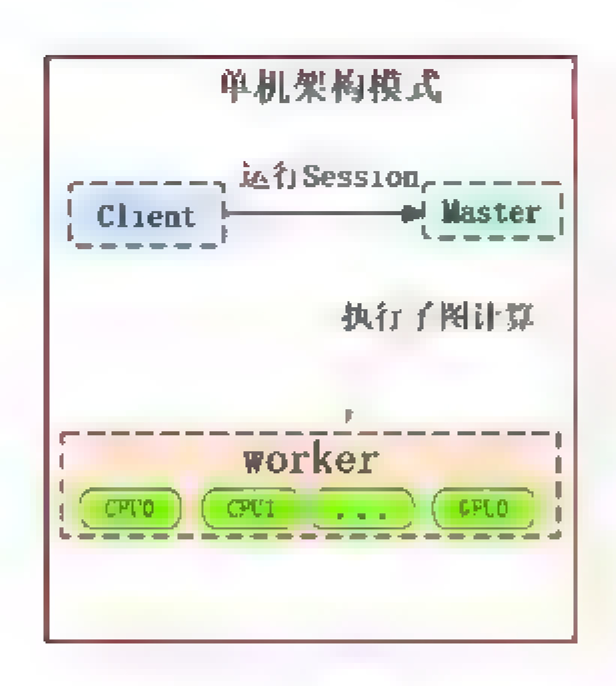
如前所述，TensorFlow 中的会话（Session）用来执行事先定义好的运算，负责完成多个计算设备或集群分布式节点的布置和数据传输节点的添加，并且还负责将子图分配给相应的执行器单元进行运行。会话拥有并管理 TensorFlow 程序运行时的所有资源。当计算完成之后需要关闭会话来帮助系统回收资源，否则就可能出现计算资源被占用的问题。

3.6 TensorFlow 工作原理

TensorFlow 是一个具有“client—master—worker”架构的分布式系统。在 TensorFlow 中，参与分布式系统的所有节点或者设备被统称为一个 cluster（集群），一个 cluster 中包含多个 server（服务器），每个 server 去执行一项 task（任务），而 server 和 task 又是一一对应的。这样看来，我们可以把 cluster 看成是 server 的集合，也可以看成是 task 的集合。TensorFlow 为每个 task 又增加了一个抽象层，将一系列相似的 task 集合称为一个 job（工作）。例如，我们在 PS 架构中，习惯称 parameter server（参数服务器）的 task 集合为 ps，而把执行梯度计算的 task 集合称为 worker。因此，cluster 又可以被看成是 job 的集合，实际上这只是逻辑上的意义，我们还需要具体看一下这个 server 真正做的是什么。在 TensorFlow 中，job 用 name（字符串）标识，task 用 index（整数索引或整数下标）标识，cluster 中的每个 task 都可以用 job 的 name 加上 task 的 index 来作唯一标识。在分布式系统中，一般情况下各个 task 在不同的节点或者设备上执行。

通常情况下，客户端（client）通过会话 tf.Session 接口和 master 展开通信，且将触发执行的请求提交给 master，而 master 会把所要执行的任务分派给单个或多个 worker 进程，相关结果通过 master 返回给客户端。这里，worker 负责计算的执行，任何一个 worker 进程都会管理和使用计算机上的计算硬件设备资源，比如一块或者多块 CPU 和 GPU，进而处理计算子图（Subgraph）的运算或操作过程。

其实，TensorFlow 的架构横向扩展是很灵活的。在单机模式的多数情况下，client、master 和 worker 均在同一个进程中启动，而 worker 进程会管理本机上所有的计算设备资源。在分布式的架构中，我们可以通过远程调用的方式将 client、master 和 worker 连接在一起，且任何一个 worker 进程各自监控关联执行机上的计算设备。关于 TensorFlow 的单机和分布式架构如图 3-4 所示。



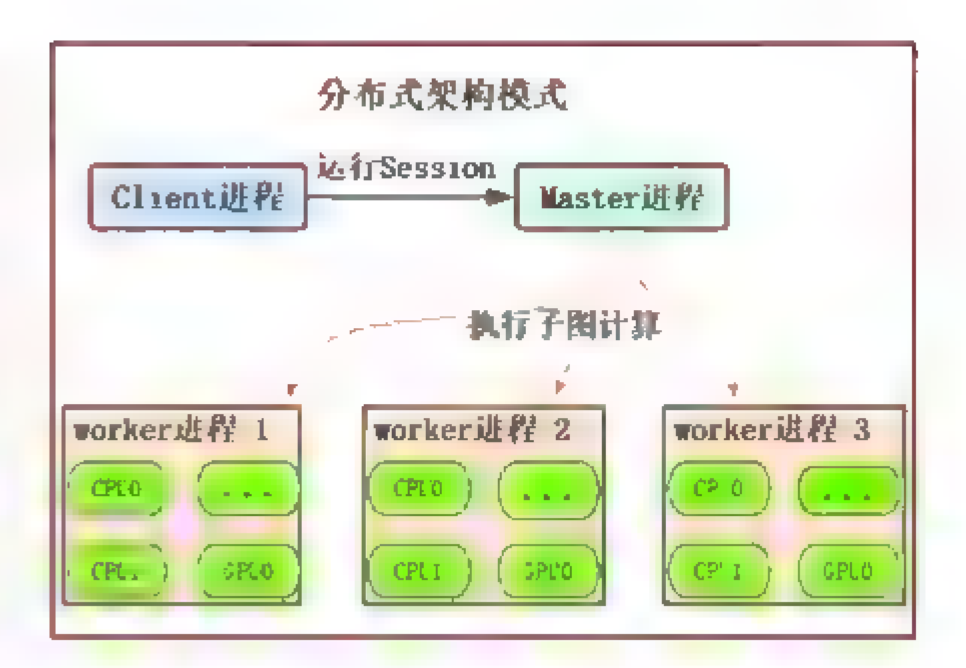


图 3-4　TensorFlow 的单机和分布式架构示意图

在创建计算图之后，TensorFlow 需要以分布式方式由多个 CPU（以及 GPU，如果可用）执行激活的会话。通常，实际上不需要明确指定是使用 CPU 还是 GPU，因为 TensorFlow 可以选择并使用其中一个。默认情况下，将选择 GPU 进行尽可能多的运算；否则，将使用 CPU。因此，从广义上看，以下是 TensorFlow 的主要组件：

- 变量（Variable）：用于包含 TensorFlow 会话之间权重值和偏差的值。
- 张量（Tensor）：在节点之间传递的一组值。
- 占位符（Placeholder）：用于在程序和 TensorFlow 图之间发送数据。
- 会话（Session）：启动会话时，TensorFlow 会自动计算图中所有计算的梯度，并在链式规则中使用它们。实际上，在执行图形时会调用会话。

从技术上说，我们要编写的程序可以被视为客户端，而客户端用于以符号方式在 C/C ++或 Python 中创建计算图，接着，我们的代码可以要求 TensorFlow 执行此该图，请参见图 3-5 中的详细信息。



图 3-5　在 TensorFlow 分布式架构下执行代码示意图

计算图有助于在具有 CPU 或 GPU 的多个计算节点上分配工作负载。这样，神经网络等同于复

合函数，其中每个层（输入层、隐藏层或输出层）可以表示为函数。现在想要了解在张量上执行的运算或操作，有必要探讨 TensorFlow 编程模块方面的解决方案。

3.7 通过一个示例来认识 TensorFlow

现在让我们结合示例对 TensorFlow 框架中的一些基本组件进行更多的解读。我们编写一个示例来执行以下计算，这在神经网络中非常常见：

$$h = \text{sigmoid}(W * x + b)$$

这里 W 和 x 是矩阵，b 是向量，*表示点积。 sigmoid 是一个非线性变换，由下式给出：

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

我们将逐步讨论如何通过 TensorFlow 进行上述计算。

首先，我们需要导入 TensorFlow 和 NumPy。在 Python 中运行任何类型的 TensorFlow 或 NumPy 相关操作之前，均必须导入它们：

```
import tensorflow as tf
import numpy as np
```

接下来，我们定义一个图对象，稍后将使用运算或操作和变量填充它们：

```
graph = tf.Graph() # 创建一个图（graph）对象
session = tf.InteractiveSession(graph=graph) # 创建一个会话（session）对象
```

graph 对象包含一个计算图，该计算图连接我们在程序中定义的各种输入和输出，以获得最终所需的输出（即它定义 W、x 和 b 如何被连接以便我们根据图生成 h）。此外，我们将定义一个会话对象，该对象以定义的图作为输入，执行该图。我们后面会详细讨论这些元素。

要创建新的图对象，可以使用以下操作：

```
graph = tf.Graph()
```

也可以使用以下操作来获取 TensorFlow 默认计算图：

```
graph = tf.get_default_graph()
```

现在我们将定义一些张量，即 x、W、b 和 h。在 TensorFlow 中，定义张量有几种不同的方法，在这里列出三种：

（1）首先，x 是一个占位符。顾名思义，占位符没有被初始化。相反，我们将在图执行时提供一些数值。

（2）其次，这里有变量 W 和 b。由于变量是可变的，这意味着它们的值可以随时间而变化。

（3）最后，我们有 h，这是一个通过对 x、W 和 b 执行一些操作而产生的不可变的张量：

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W')
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
h = tf.nn.sigmoid(tf.matmul(x,W) + b)
```

注意，对于 W 和 b，我们提供了一些重要的参数：

```
tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,dtype=tf.float32)
tf.zeros(shape=[5],dtype=tf.float32)
```

这些被称为变量初始化器，是最初分配给 W 和 b 变量的张量。变量不能在没有初始值的情况下作为占位符流动，并且需要始终为它们分配一些数值。这里，tf.random_uniform 意味着我们在 minval(-0.1)和 maxval(0.1)之间均匀地采样以将值赋给张量，并且 tf.zeros 用零初始化张量。在定义张量时，定义张量的 shape 也非常重要。shape 属性定义张量每个维度的大小。例如，如果形状（shape）是[10,5]，这意味着它将是一个二维结构，在 0 轴上有 10 个元素，在 1 轴上有 5 个元素。

接下来，我们将进行初始化操作，初始化图中的变量 W 和 b：

```
tf.global_variables_initializer().run()
```

现在，将执行计算图以获得我们需要的最终输出 h。这是通过运行 session.run(...)来完成的，我们将以占位符的值作为 session.run()命令的参数：

```
h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
```

最后，关闭会话，释放 session 对象持有的所有资源。

```
session.close()
```

以下是此 TensorFlow 示例的完整代码。本章中的所有代码示例都将在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件中提供：

```
import tensorflow as tf
import numpy as np
# 定义 graph 和 session
graph = tf.Graph() # Creates a graph
session = tf.InteractiveSession(graph=graph) # Creates a session

# 构建 graph
# 占位符是一种符号输入
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W') # 相关变量

# 相关变量
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
```

```
h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to be performed
# 在图中执行操作和评估节点
tf.global_variables_initializer().run() # 初始化变量
#通过为输入 x 提供数值来运行该操作
h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
# 关闭会话以释放会话中的所有已保存资源
session.close()
```

3.8 TensorFlow 客户端

前面的示例程序可以称为 TensorFlow 客户端，本节结合示例和上面的内容，对 TensorFlow 客户端做详细的解读。在使用 TensorFlow 编写的任何客户端程序中，主要将有两种主要类型的对象：运算（Operation）和张量（Tensor）。在前面的例子中，tf.nn.sigmoid 是一个运算，h 是张量。

然后，这里还有一个 graph 对象，它是存储程序数据流的计算图。当我们在代码中添加定义的 x、W、b 和 h 的后续代码行时，TensorFlow 会自动将这些张量（Tensor）和每个运算（Operation）（例如，tf.matmul()）作为节点添加到图中。该图将存储重要信息，例如张量的依赖性以及要执行的运算。在我们的示例中，如果要计算 h，那么张量 x、W 和 b 是必需的。因此，如果在运行时没有正确初始化其中一个参数，TensorFlow 将在这里指出需要修正的具体的初始化错误。

接着，由前面内容，我们知道会话（Session）将扮演执行计算图的角色，方法是将图划分为子图、更精细的图，然后将这些图分配给将执行指定任务的 worker，这是通过 session.run(...)函数来完成的。下面，我们来看看 TensorFlow 架构中执行客户端时的情况。

大家知道，TensorFlow 擅长创建一个包含所有依赖关系和运算的精确计算图，且它能够准确地知道数据流以何种方式、何时、在哪里流动。当然，还有一个元素使得 TensorFlow 变得更优秀，那就是能够有效执行计算图的会话（Session），这也是会话进入 TensorFlow 架构的入口之处。现在让我们来看看会话的内部情况，以了解图形是如何被执行的。

首先，由前面的介绍，我们知道 TensorFlow 客户端中的相关元素有计算图、张量和会话。创建会话时，它会将计算图作为 tf.GraphDef 协议缓冲区发送到分布式架构的主机。 tf.GraphDef 是图的标准化表示。分布式主服务器查看图中的所有计算，并将计算划分到不同的设备（例如，不同的 GPU 和 CPU）。我们的 sigmoid 示例中的计算图如图 3-6 所示。

接下来，计算图将被分解为子图，并进一步细分为更细的部分，这在具有许多隐藏层的现实世界解决方案中意义更大。此外，为了并行地执行任务（例如，多个设备），将计算图分解为多个部分就变得很重要。执行此图（若该图被划分为子图，则执行子图）称为单个任务，其中任务被分配给单个 TensorFlow 服务器。

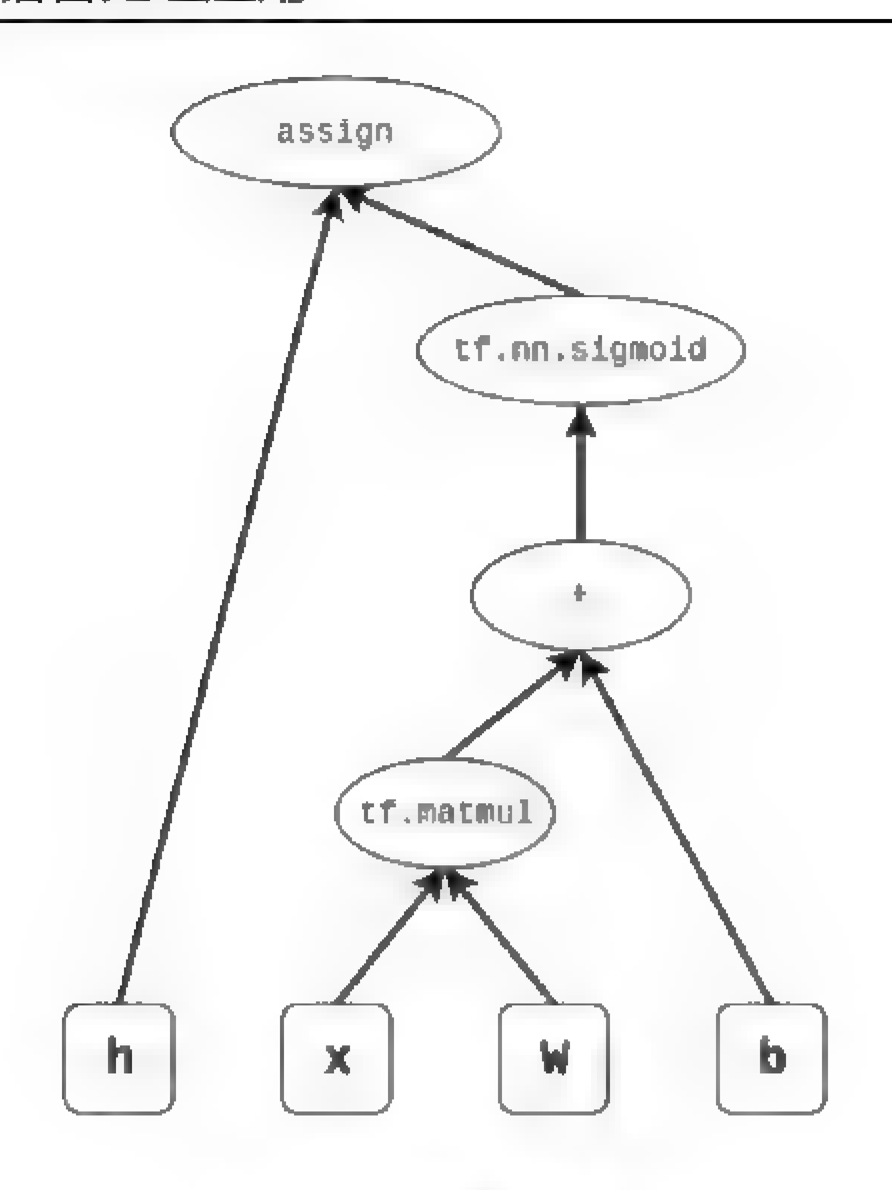


图 3-6 客户端的计算图

然而，在现实中，每个任务都是通过将其分解为两个部分来执行的，每个部分都是由单个 worker 执行的：

- 一个 worker 使用参数的当前值（运算执行器）执行 TensorFlow 操作。
- 一个 worker 存储参数，并使用执行运算器后获得新值来更新它们（参数服务器）。

TensorFlow 客户端的通用工作流程如图 3-7 所示。

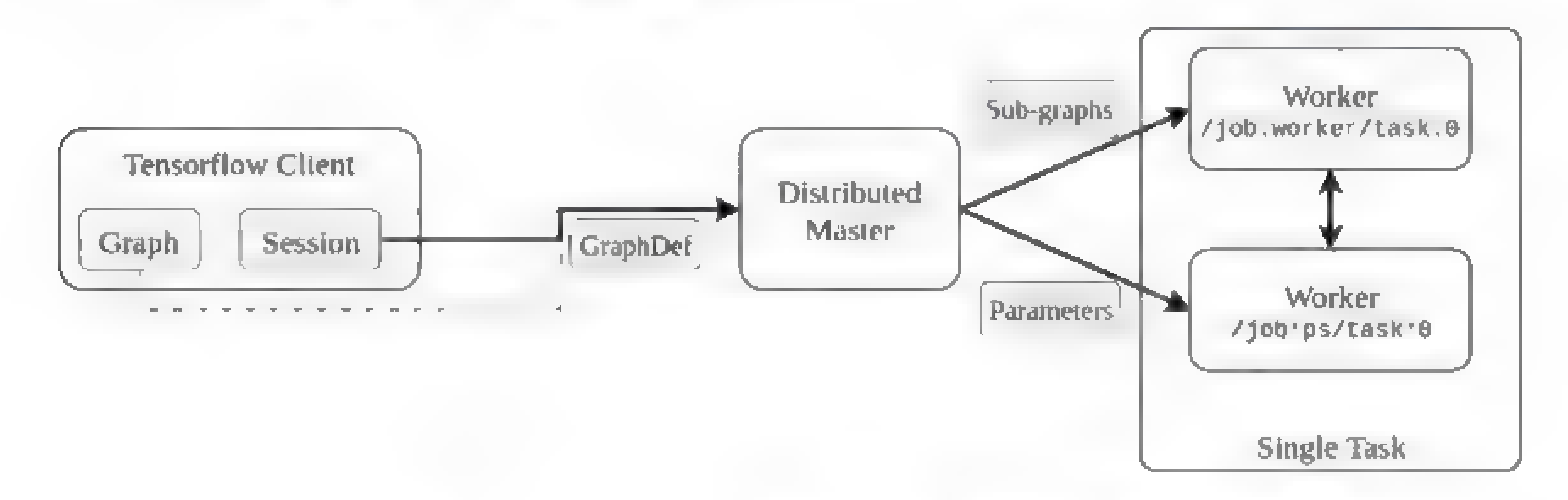


图 3-7 TensorFlow 客户端的通用执行路线图

图 3-8 说明了计算图的分解情况。除了分解计算图之外，TensorFlow 还插入发送节点和接收节点，以利于参数服务器和运算执行器之间的通信。我们可以理解为每当数据可用时，发送节点就发送数据，其中接收节点在对应的发送节点发送数据时继续侦听和捕获数据。

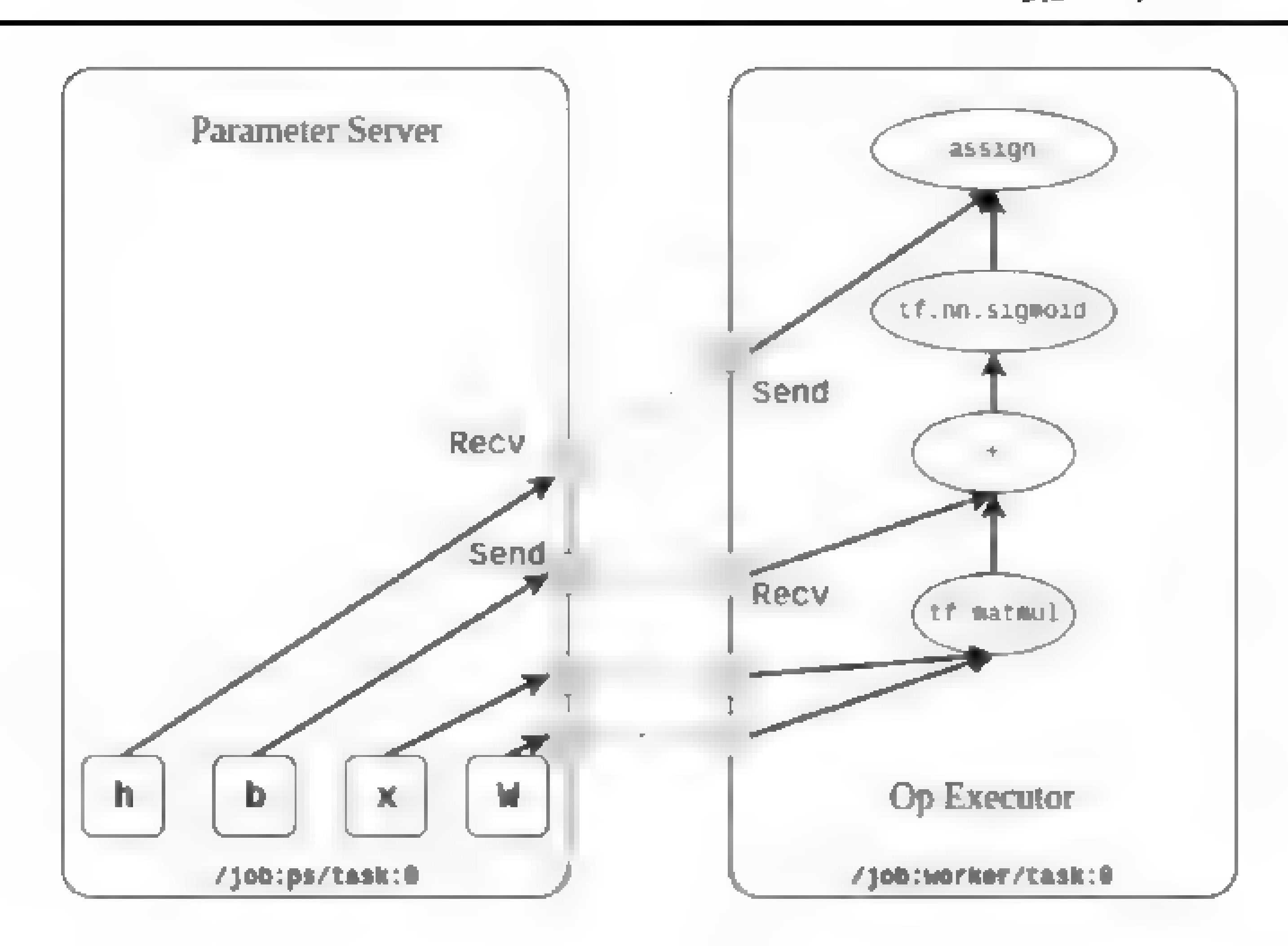


图 3-8 计算图的分解情况

最后，一旦计算完成，会话就将更新的数据从参数服务器端带回到客户端。从现在技术角度来看，我们也可以给出 TensorFlow 的技术架构，如图 3-9 所示。此解释基于 https://tensorflow.google.cn/extend/architecture 上的官方 TensorFlow 文档。

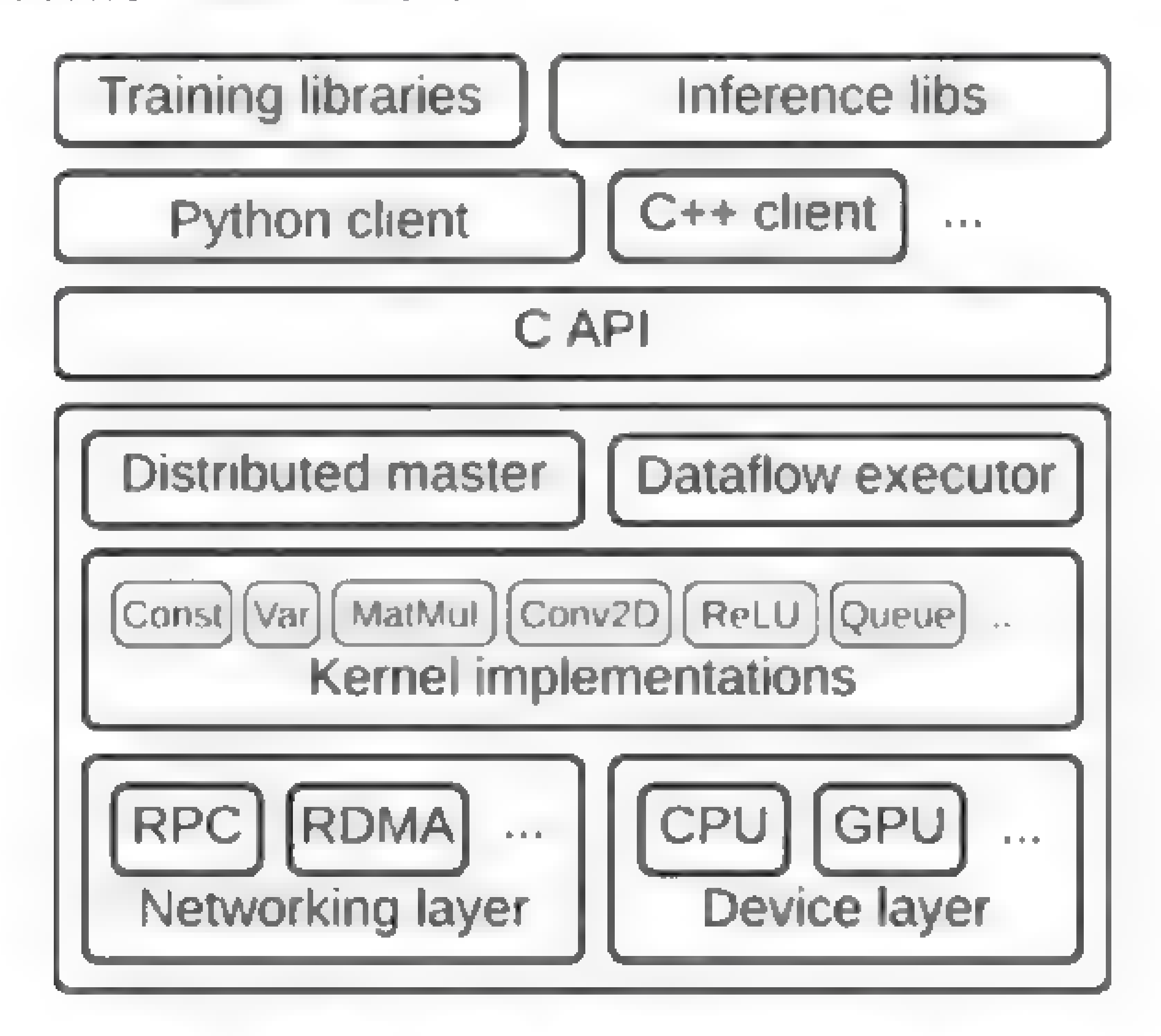


图 3-9 TensorFlow 技术架构

3.9 TensorFlow 中常见元素解读

在了解底层架构的同时，我们下面将介绍构成 TensorFlow 客户端中最常见的元素。

- **输入（Input）：** 用于训练和测试算法的数据。

- **变量（Variable）**：可变张量，主要定义算法的参数。
- **输出（Output）**：存储终端和中间输出的不可变张量。
- **运算或操作（Operation）**：输入的各种变换以产生期望的输出。

在之前的 sigmoid 示例中，我们可以找到相关类别的实例，如表 3-2 中所示。

表 3-2 sigmoid 例子中相关类别的实例

TensorFlow 元素	示例客户端的值
Input—输入	x
Variable—变量	W 和 b
Output—输出	h
Operation—运算操作	tf.matmul(...), tf.nn.sigmoid(...)

接下来，我们将详细解读 TensorFlow 中的这些元素。

3.9.1 在 TensorFlow 中定义输入

客户端主要以下面三种不同的方式接收数据：

- 使用 Python 代码在算法的每个步骤中提供数据。
- 将数据预加载并存储为 TensorFlow 张量。
- 构建输入管道。

接下来，具体看看这三种不同的方式。

1. 使用 Python 代码提供数据

在第一种方法中，可以使用传统的 Python 代码方法将数据提供给 TensorFlow 客户端。在我们之前的示例中，x 是此方法的示例。为了从外部数据结构（例如，numpy.ndarray）向客户端提供数据，TensorFlow 库提供了一种极佳的数据结构符号，称为占位符（Placeholder），定义为 tf.placeholder(...)。前面我们说过，占位符在计算图构建阶段不需要实际数据，相反，我们仅通过执行 session.run(..., feed_dict = {placeholder: value})调用的图时提供数据，方法是将外部数据以 Python 字典的形式传递给 feed_dict 参数。占位符的定义如下：

```
tf.placeholder(dtype, shape=None, name=None)
```

参数如下：

- dtype：这是提供占位符的数据的类型。
- shape：这是占位符的 shape，以一维向量给出。
- name：这是占位符的名称，对于调试很重要。

2. 将数据预加载并存储为张量

第二种方法与第一种方法类似，但有一点需要注意。我们不必在计算图执行期间提供数据，因为数据是预先加载的。为了了解这一点，让我们修改一下 sigmoid 示例，将 x 定义为占位符：

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
```

另外，也将 x 定义为包含具体数值的张量：

```
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')
```

其完整代码将变为如下：

```
import tensorflow as tf
# 定义 graph 和 session
graph = tf.Graph()

session = tf.InteractiveSession(graph=graph)
# 创建计算图 graph
#x - 预加载的输入
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,
dtype=tf.float32),name='W') # 变量
# 相关变量
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
        h = tf.nn.sigmoid(tf.matmul(x,W) + b) # 将要执行的运算
        # 在图中执行运算和评估节点
        tf.global_variables_initializer().run() # 初始化变量
        #执行不带有 feed_dict 的运算
        h_eval = session.run(h)
        print(h_eval)
        #关闭会话，释放资源
        session.close()
```

其实，可以发现这里与原来 sigmoid 示例存在两个主要区别。我们以不同的方式定义了 x。现在直接指定一个具体值并将 x 定义为张量，而不是使用占位符对象并在计算图执行时输入实际值。另外，正如你看到的，我们在 session.run(...)中不会提供任何额外的参数。但是，在缺点方面，现在我们无法在 session.run(..)处向 x 提供不同的值并查看输出是如何变化的。

3. 构建数据输入管道

输入管道是专门为需要快速处理大量数据的“重量级”客户端而设计的。这实际上创建了一个保存数据的队列，直到等到需要它的时候为止。TensorFlow 还提供了各种预处理步骤（例如，用于调整图像对比度/亮度或者标准化），在将数据提供给算法之前执行。为了提高效率，可以让多

个线程并行读取和处理数据。

（1）数据输入管道的结构

TensorFlow 数据输入管道也可以被抽象为一个 ETL 过程（Extract、Transform、Load）。

- Extract：从硬盘上读取数据，可以是本地（HDD 或 SSD），也可以是网盘（GCS 或 HDFS）。
- Transform：使用 CPU 去解析、预处理数据，比如图像解码、数据增强、变换（比如随机裁剪、翻转、颜色变换）、打乱（shuffle）、分批量（batching）。
- Load：将 Transform 后的数据加载到计算设备，例如 GPU、TPU 等设备。

这种模式有效地利用了 CPU，从而让 GPU、TPU 等设备专注于进行模型的训练过程（提高了设备的利用率）。

具体来看，典型的管道将包含以下组件：

- 文件名列表。
- 文件名队列，为输入（记录）阅读器生成文件名。
- 用于读取输入的记录阅读器（记录）。
- 解码读取记录的解码器（例如，JPEG 图像解码）。
- 预处理步骤（可选）。
- 一个示例（解码输入）队列。

（2）数据输入管道（Pipeline）的优化

随着新的计算设备（例如 GPU 和 TPU）的应用，使得以越来越快的速度训练神经网络成为可能，CPU 处理方式很容易遇到海量数据计算的瓶颈。tf.data API 提供数据输入管道所需的各种部件，可以对数据输入管道进行优化。

在执行一个训练 step 之前，必须先做 Extract、Transform 训练数据，然后将其提供给计算设备上运行的模型。在以前，当 CPU 在准备数据时，计算设备处于闲置状态；当计算设备执行训练模型时，CPU 又处于闲置状态。因此，单个训练 step 的时间等于 CPU 准备数据的时间和计算设备执行训练 step 时间的总和。

Pipelining 操作将训练 step 中的数据准备和模型执行实现了并行操作。当计算设备在执行第 N 个训练 step 时，CPU 将为第 N+1 个训练 step 准备数据。这样，通过两个过程的重叠，单个训练 step 的时间将取 CPU 准备数据的时间和计算设备执行训练 step 的时间中的较大值。

如果我们没有进行管道化（Pipelining）操作，CPU 和 GPU/TPU 中大部分时间处于空闲状态，如图 3-10 所示。

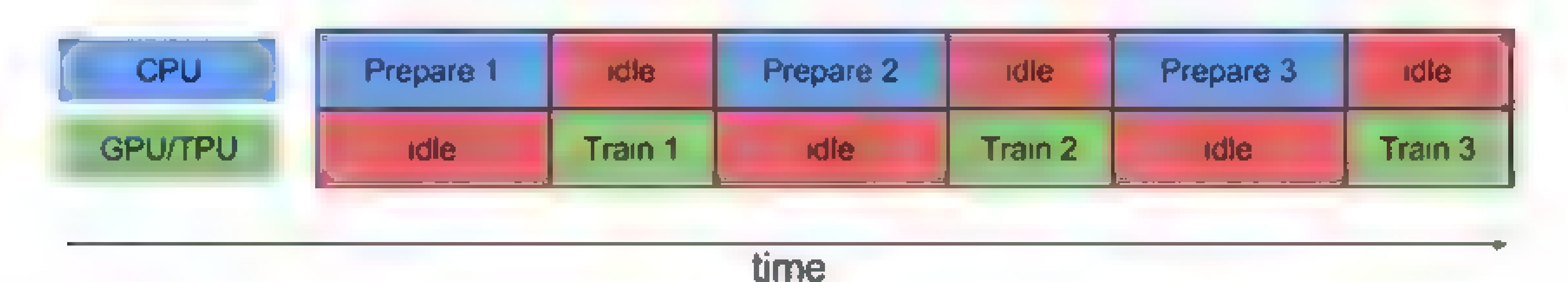


图 3-10　没有使用 Pipelining 操作的 CPU 和 GPU / TPU 处于空闲状态的时间示意图

我们如果使用 Pipelining 操作，CPU 和 GPU/TPU 中处于空闲状态的时间显著减少，如图 3-11 所示。

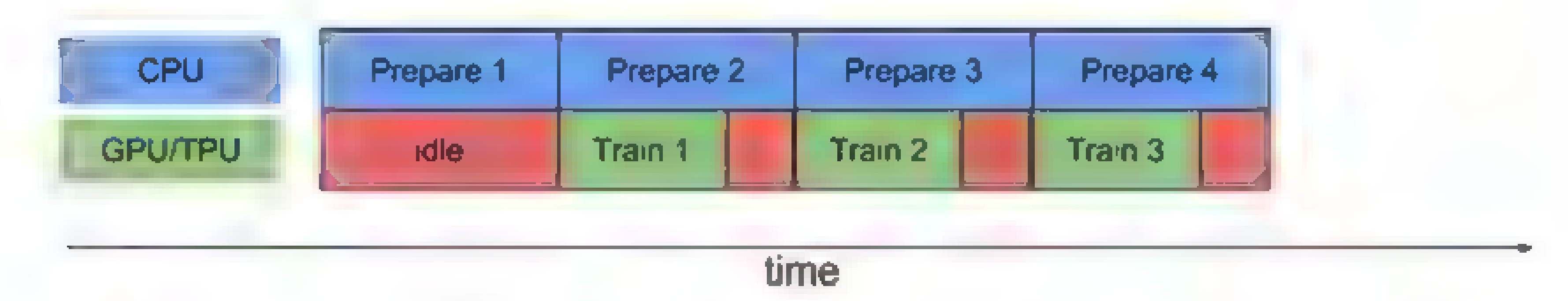


图 3-11　使用 Pipelining 操作的 CPU 和 GPU/TPU 处于空闲状态的时间示意图

提示

有关更多信息，请参阅 https://tensorflow.google.cn/guide/performance/datasets#input_pipeline_structure 上有关数据输入管道的官方说明。

（3）编写输入管道示例

让我们使用 TensorFlow 编写一个输入管道示例。在这个例子中，我们有三个 CSV 格式的文本文件（text1.txt，text2.txt 和 text3.txt），每个文件有 5 行，每行有 10 个用逗号分隔的数字（示例行：0.1、0.2、0.3、0.4、0.5、0.6、0.7、0.8、0.9、1.0）。接着，我们具体来看一下。

提示

更多相关信息，请参阅 https://tensorflow.google.cn/programmers_guide/reading_data 上有关导入数据的 TensorFlow 官方说明。

首先，和以前一样导入一些重要的库：

```
import tensorflow as tf
import numpy as np
```

接下来，我们将定义计算图 graph 和会话 session 对象：

```
graph = tf.Graph()
session = tf.InteractiveSession(graph=graph)
```

然后，我们将定义一个文件名队列，一个包含文件名的队列数据结构。这将作为参数传递给 reader（后面将被定义）。队列将根据 reader 的请求生成文件名，以便读者可以使用这些文件名获取文件进而读取数据：

```
filenames = ['test%d.txt'%i for i in range(1,4)]
filename_queue = tf.train.string_input_producer(filenames, capacity=3,

shuffle=True,name='string_input_producer')
```

这里，capacity 是在给定时间时队列中保存的数据量，shuffle 告诉队列是否在发出数据之前对数据进行重新打乱（shuffle）。

TensorFlow 有几种不同类型的 reader（https://tensorflow.google.cn/api_guides/python/io_ops#Readers 上提供了可用 reader 列表）。由于我们有一些单独的文本文件，其中一行代表一个数据点，因此，这里 TextLineReader 是最适合的：

```
reader = tf.TextLineReader()
```

在定义 reader 之后，我们可以使用 read()函数从文件中读取数据。它输出的是“键-值”对。该键识别出文件和该文件中正在读取的记录（文本行），我们可以省略这个。该值返回 reader 所读取行的实际值：

```
key, value = reader.read(filename_queue, name='text_read_op')
```

下面我们将定义 record_defaults，如果发现存在任何错误记录提示，将给出如下输出：

```
record_defaults=[[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],[-1.0], [-1.0],
[-1.0], [-1.0]]
```

现在，我们将读取的文本行解码为数字列（就像我们的 CSV 文件一样）。为此，我们使用 decode_csv()方法。打开文件（例如，test1.txt），将看到在一行中有 10 列：

```
col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 =tf.decode_csv(value,
record_defaults=record_defaults)
```

然后，我们将连接这些列来组成单个张量（称之为特征），这些张量将被传递给另一个方法 tf.train.shuffle_batch()。tf.train.shuffle_batch()方法采用先前定义的张量，随机填充张量并输出一批给定的批量大小：

```
features=tf.stack([col1, col2, col3, col4, col5, col6, col7, col8,col9, col10])
x = tf.train.shuffle_batch([features], batch_size=3, capacity=5,
                                   name='data_batch', min_after_dequeue=1,
                                         num_threads=1)
```

batch_size 参数是我们在给定 step 中采样的数据批量大小，capacity 是数据队列的容量（大型队列需要更多内存），min_after_dequeue 表示部分元素出队后还留在队列中的最小元素数量。最后，num_threads 定义了用于生成一批数据的线程数。如果管道中进行了大量预处理，可以增加这个线程数量。此外，如果我们需要在不进行 shuffle 的情况下读取数据（与 tf.train.shuffle_batch 一样），就可以调用 tf.train.batch 方法。接着，我们将通过以下命令来启动此管道：

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(coord=coord, sess=session)
```

可以将 tf.train.Coordinator()类看成为线程管理器。它控制着各种管理线程的机制。我们需要 f.train.Coordinator()类，因为输入管道产生许多线程来填充入队队列、出列队列以及许多其他任务。接下来，我们将使用之前创建的线程管理器去执行 tf.train.start_queue_runners(...)。QueueRunner()保存队列的入队操作，这些操作是在定义输入管道时自动创建的。因此，要填充已定义的队列，我

们需要调用 tf.train.start queue runners 函数来启动这些队列运行的程序。

接下来，在指定任务完成之后，我们需要停止相关线程并将它们连接到主线程，否则眼前的程序将无限期挂起。这是通过调用 coord.request stop()和 coord.join(threads)来实现的。这个数据输入管道与我们的 sigmoid 示例相结合，便能够直接从文件中读取数据，如下所示：

```
    import tensorflow as tf
    import numpy as np
    import os
    # 定义 graph 和 session
    graph = tf.Graph()
    session = tf.InteractiveSession(graph=graph)
    ### 创建数据输入管道 ###
    # 文件名队列
    filenames = ['test%d.txt'%i for i in range(1,4)]
    filename_queue = tf.train.string_input_producer(filenames, capacity=3,
    shuffle=True,name='string_input_producer')
    # 检查所有文件是否存在
    for f in filenames:
    if not tf.gfile.Exists(f):
       raise ValueError('Failed to find file: ' + f)

    else:
       print('File %s found.'%f)
    #reader 接受一个文件名队列和 read()函数，read()函数依次输出数据
    reader = tf.TextLineReader()
    # 输出"键-值"对
    key, value = reader.read(filename_queue, name='text_read_op')
    # 如果在读取文件时遇到任何问题，这是返回的值
    record_defaults=[[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],[-1.0], [-1.0],
[-1.0], [-1.0]]
    # 将读取到的数值解码为列
    col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 =
    tf.decode_csv(value, record_defaults=record_defaults)

    # 现在我们将这些列叠加在一起，形成一个包含所有列的单个张量
    features=tf.stack([col1, col2, col3, col4, col5, col6, col7, col8,col9, col10])
    # 输出 x 被随机分配一批 batch_size 数据，其中从.txt 文件中读取数据
    x = tf.train.shuffle_batch([features], batch_size=3, capacity=5,
name='data_batch',min_after_dequeue=1,num_threads=1)
    #QueueRunner 从队列中检索数据，我们需要显式地启动它们
    # Coordinator 协调多个 QueueRunners
    coord = tf.train.Coordinator()
    threads = tf.train.start queue runners(coord coord, sess session)
```

```
# 通过定义变量和计算来构建计算图
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W') # 变量
# 相关变量
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
h = tf.nn.sigmoid(tf.matmul(x,W) + b) # 要执行的运算
#在图中执行运算和评估各节点
tf.global_variables_initializer().run() # 初始化变量
# 用 x 计算 h 并打印 5 个步的结果
for step in range(5):
x_eval, h_eval = session.run([x,h])
print('========== Step %d =========='%step)
print('Evaluated data (x)')
print(x_eval)
print('Evaluated data (h)')
print(h_eval)
print('')
# 关闭 coordinator，否则程序将无限期挂起
coord.request_stop()
coord.join(threads)
session.close()
```

3.9.2 在 TensorFlow 中定义变量

变量在 TensorFlow 中扮演着重要角色。变量本质上是一个张量，具有特定的形状（shape），该形状（shape）用于定义变量将具有多少个维度以及每个维度的大小。然而，与常规张量不同，变量是可变的，也就意味着变量的值在定义后是可以改变的。这是实现学习模型参数（例如，神经网络权重值）的理想属性，其中权重值在每个学习步骤之后会稍有变化。例如，如果使用 x = tf.Variable(0,dtype=tf.int32)定义变量，则可以使用诸如 tf.assign(x,x+1)之类的 TensorFlow 运算来更改该变量的值。但是，如果这样定义张量，例如 x = tf.constant(0，dtype = tf.int32)，就无法更改该张量的值，它应该保持 0 的状态直到程序执行结束。

变量的创建很简单。在我们 sigmoid 的示例中，已经创建了两个变量 W 和 b。在创建变量时，有一些事情非常重要，在这里列出它们并在后续中详细讨论：

- Variable shape：变量形状。
- Data type：数据类型。
- Initial value：初始值。
- Name (optional)：名称（可选）。

变量形状是[x，y，z，...]格式的一维向量。列表中的每个值表示相应维度或轴的大小。例如，如果需要具有 50 行和 10 列的二维张量作为变量，那其形状将是 [50,10]。

变量的维数（形状向量的长度）在 TensorFlow 中被识别为张量的秩。不要把它与矩阵的秩混淆起来。

提示

TensorFlow 中张量的秩表示张量的维数；但对于二维矩阵来说，其秩等于 2。

其实，数据类型在确定变量大小方面起着重要作用。有许多不同的数据类型，包括常用的 tf.bool、tf.uint8、tf.float32 和 tf.int32（代码中的字段类型，其意义基本上都是通用的，这和我们平时编写 Java 或.net 程序时用到的布尔类型、浮点类型、整数类型类似，而 unit8 是 8 位无符号整型）。每种数据类型都具有属于该类型的单个值所需的位数(bit)。例如，tf.uint8 类型需要有 8 位，而 tf.float32 需要 32 位。通常的做法是使用相同的数据类型进行计算，否则会导致数据类型的不匹配。因此，如果需要对两个具有不同数据类型的张量进行转换，需要使用 tf.cast(...)操作将一个张量显式地转换为另一个张量的类型。例如，对于具有 tf.int32 数据类型的 x 变量，需要将其转换为 tf.float32 的类型，tf.cast(x，dtype=tf.float32)的调用就是将 x 变量转换为 tf.float32 类型。

接下来，我们需要对变量进行初始化，TensorFlow 也提供了几种不同的初始化器，包括常数初始化器和正态分布初始化器。 TensorFlow 一些主要的初始化器如下所示：

- tf.zeros
- tf.constant_initializer
- tf.random_uniform
- tf.truncated_normal

最后，变量的名称（name）将作为一个标识符（ID），使我们能够在图（Graph）中对该变量进行识别。因此，如果我们对计算图进行可视化操作，那变量将通过传递 name 关键字参数的形式进行显示。如果未指定名称，TensorFlow 将使用默认命名方案。

注意

Python 变量 tf.variable 是赋给计算图的，它不是 TensorFlow 变量命名的一部分。考虑下面的示例，可以在其中指定 TensorFlow 变量：

```
a = tf.Variable(tf.zeros([5]), name ='b')
```

这里，TensorFlow 的图将通过名称 b 而不是 a 来识别该变量。

3.9.3 定义 TensorFlow 输出

TensorFlow 的输出通常是张量，可以是输入或变量或两者转换后的结果。在我们的例子中，h 是一个输出，其中 h = tf.nn.sigmoid（tf.matmul（x，W）+ b）。当然，有时候 TensorFlow 的一个输出也可能变成下一个输入的内容，依次输出下去可以形成一组链式运算或操作，而这里的运算或操作也不一定必须是 TensorFlow 运算或操作，还可以使用标准 Python 算法，例如：

```
x = tf.matmul(w,A)
y = x + B
z = tf.add(y,C)
```

3.9.4 定义 TensorFlow 运算或操作

在 https://tensorflow.google.cn/api_docs/python/ 上查看 TensorFlow API，你会发现 TensorFlow 有大量可用的运算或操作。下面，我们将对 TensorFlow 中几个常见的运算或操作进行解读。

1. 比较运算

比较运算对于比较两个张量非常有用。对于比较运算部分的详细资料，读者可以查阅 https://github.com/tensorflow/docs/tree/master/site/en/api_guides/python 中比较运算符的详细内容。当然，对于比较运算工作原理的理解，通过代码层面可能会更直观些。这里，我们给出示例张量 x 和 y：

```
#假设 x 和 y 的取值如下
#x (2-D tensor) => [[1,2],[3,4]]
#y (2-D tensor) => [[4,3],[3,2]]
x= tf.constant([[1,2],[3,4]], dtype=tf.int32)
y= tf.constant([[4,3],[3,2]], dtype=tf.int32)

# 检查两个张量在元素方面是否相等，并返回布尔类型的张量
# x_equal_y => [[False,False],[True,False]]
x_equal_y = tf.equal(x, y, name=None)
#检查 x 在对应元素方面是否小于 y 并返回布尔类型的张量
# x_less_y => [[True,True],[False,False]]
x_less_y = tf.less(x, y, name=None)
# 检查 x 在对应元素方面是否大于或等于 y 并返回布尔类型的张量
# x_great_equal_y => [[False,False],[True,True]]

x_great_equal_y = tf.greater_equal(x, y, name=None)
# 从 x 和 y 中选择元素，具体取决于条件是否满足（从 x 中选择元素）或
# 条件失败（从 y 中选择元素）
condition = tf.constant([[True,False],[True,False]],dtype=tf.bool)
# x_cond_y => [[1,3],[3,2]]
x_cond_y = tf.where(condition, x, y, name=None)
```

2. 比较数学运算

TensorFlow 允许我们对从简单到复杂的张量执行数学运算。这里我们将讨论 TensorFlow 中提供的一些数学运算。完整的运算集可在 https://github.com/tensorflow/docs/tree/master/site/en/api_guides/python 上找到。

```
#假设 x 和 y 的取值如下
#x (2 D tensor) => [[1,2],[3,4]]
#y (2-D tensor) => [[4,3],[3,2]]
x= tf.constant([[1,2],[3,4]], dtype=tf.float32)
y = tf.constant([[4,3],[3,2]], dtype=tf.float32)
# 以元素方式增加两个张量 x 和 y
# x add y => [[5,5],[6,6]]
x add y = tf.add(x, y)
# 执行矩阵乘法（不是元素方面）
# x_mul_y => [[10,7],[24,17]]
x_mul_y = tf.matmul(x, y)
# 计算 x 元素的自然对数，相当于计算 ln (x)
# log_x => [[0,0.6931],[1.0986,1.3863]]
log_x = tf.log(x)
# 在指定轴上执行归约 (reduction)
# x_sum_1 => [3,7]
x_sum_1 = tf.reduce_sum(x, axis=[1], keepdims=False)
# x_sum_2 => [[4],[6]]
x_sum_2 = tf.reduce_sum(x, axis=[0], keepdims=True)

# 根据 segment_ids (同一段中具有相同 id 的项) 对张量进行分段，并计算数据的分段总和
data = tf.constant([1,2,3,4,5,6,7,8,9,10], dtype=tf.float32)
segment_ids = tf.constant([0,0,0,1,1,2,2,2,2,2 ], dtype=tf.int32)
# x_seg_sum => [6,9,40]
x_seg_sum = tf.segment_sum(data, segment_ids)
```

3. 比较分散（Scatter）和聚合（Gather）操作

随着人工智能的迅猛发展，尤其是 GPU 可编程性能的增强以及 GPGPU（General Purpose Computing on GPU，在图形处理器上进行通用计算）技术的不断发展，相关研究/技术人员也迫切希望基于流处理器模型的 GPU 也可以和 CPU 一样，在支持流程分支的同时，也能够实现对存储器进行灵活的读写操作。其实，Ian Buck 在进行早期的 GPU 通用可编程技术研究时，就发现 GPU 完成复杂计算任务时存在一个关键性的缺陷，那就是缺乏灵活的存储器操作。所以，他在后来的研究中就增加了对分散和聚合操作的支持，但是结果还是以牺牲一些性能为代价的情况下完成了整个过程。

在 GPU 中，CUDA 中分散和聚合操作实现的结构示意图与第一向量机中的很相似，分散允许将数据输出到非连续的存储器地址内，而聚合则允许从非连续的存储器地址内读取数据。因此，如果认为存储器（如 DRAM）是一个二维数组，分散则可以看作利用数组下标或索引将数据写入数组中的任意位置，即 a[i] = x，而聚合则可以看作是利用数组下标或索引从数组中的任意位置读出数据，即 x = a[i]。

下面，我们给出 CUDA 中分散（Scatter）和聚合（Gather）操作的结构示意图，如图 3-12 所示。其中，每个 ALU 可以看作是一个处理核心，通过分散/聚合操作，多个 ALU 之间可以共享存

储器，实现对任意地址数据的读写操作。

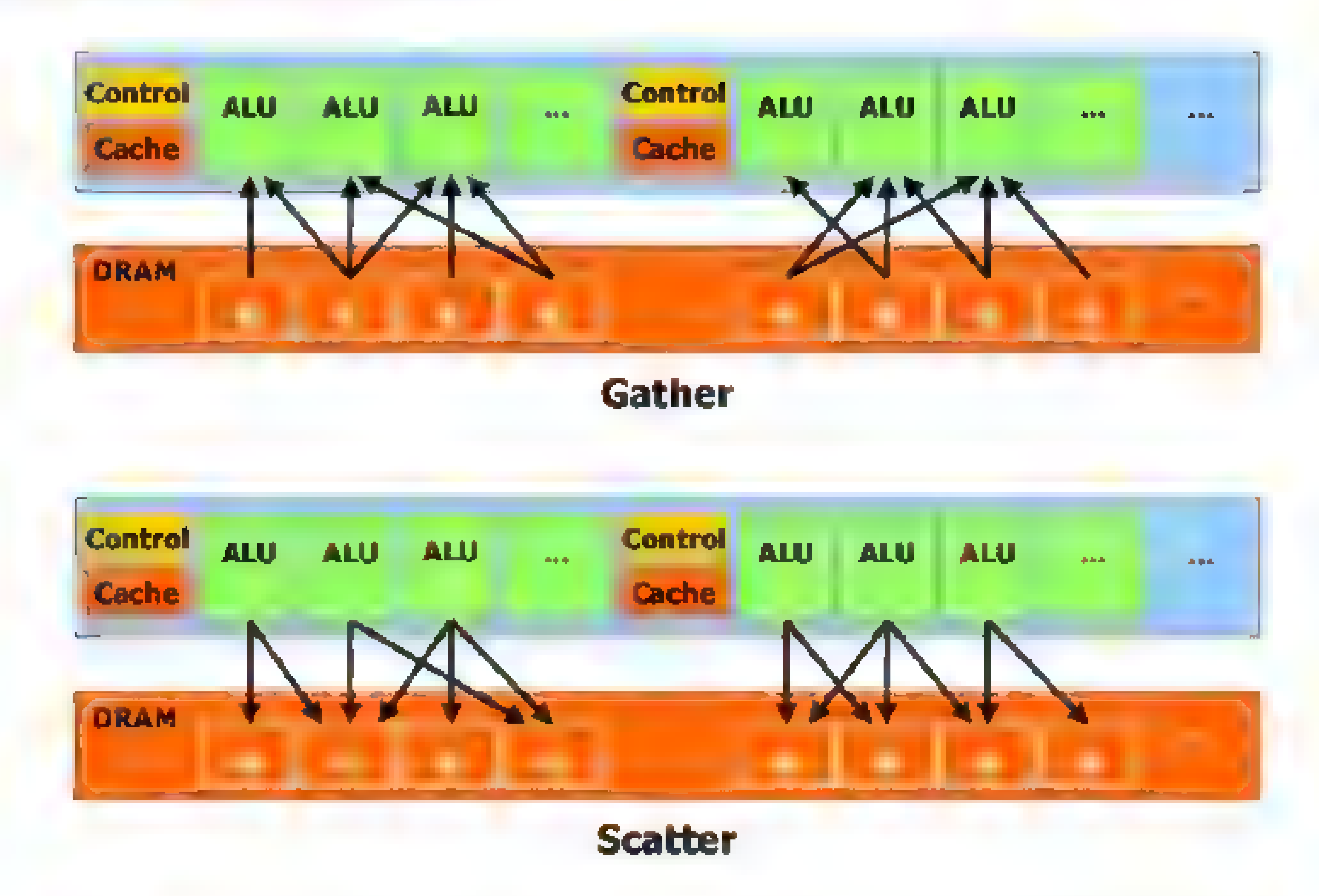


图 3-12 CUDA 中分散（Scatter）和聚合（Gather）操作的结构示意图

分散和聚合操作在矩阵运算任务中起着至关重要的作用，因为目前来看这两种变体在 TensorFlow 中是把张量编入索引的唯一方法。换句话说，不能像在 NumPy 中那样访问 TensorFlow 中的张量元素。分散操作允许我们将值分配给指定张量的特定索引，而聚合操作允许我们提取指定张量的切片（或单个元素）。以下代码显示了分散和聚合操作的一些变体：

```
    # 1-D scatter 操作
    ref = tf.Variable(tf.constant([1,9,3,10,5],dtype=tf.float32),
name='scatter_update')
    indices = [1,3]
    updates = tf.constant([2,4],dtype=tf.float32)
    tf_scatter_update = tf.scatter_update(ref, indices, updates, use_locking=None,
name=None)
            # n-D scatter 操作
            indices = [[1],[3]]
            updates = tf.constant([[1,1,1],[2,2,2]])
            shape = [4,3]
            tf_scatter_nd_1 = tf.scatter_nd(indices, updates, shape, name=None)
            # n-D scatter 操作
            indices = [[1,0],[3,1]] # 2 x 2
            updates = tf.constant([1,2]) # 2 x 1
            shape = [4,3] # 2
            tf_scatter_nd_2 = tf.scatter_nd(indices, updates, shape, name=None)
            # 1-D gather 操作
            params = tf.constant([1,2,3,4,5],dtype=tf.float32)
            indices = [1,4]
```

```
            tf gather = tf.gather(params, indices, validate indices-True,name None)
#-> [2,5]
            # n-D gather 操作
            params =
tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
            indices = [[0],[2]]
            tf gather nd = tf.gather nd(params, indices, name=None)
#=>[[0,0,0],[2,2,2]]
            params =
tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
            indices = [[0,1],[2,2]]
           tf_gather_nd_2 = tf.gather_nd(params, indices, name=None)
#=>[[0,0,0],[2,2,2]]
```

4. 比较与神经网络相关的运算或操作

下面让我们看看几个很有用的神经网络运算或操作，这些将在后面的章节中使用到。这里，我们会对简单元素的转换进行讨论，也会对一组参数对于另一个值的偏导数的运算进行讨论，并给出一个简单的神经网络实现例子。

（1）神经网络的非线性激活

非线性激活能够使神经网络很好地执行许多任务。通常，在神经网络的每一层输出后（最后一层除外）都会有一个非线性的激活转换（激活层）。非线性变换有助于神经网络学习数据中出现的各种非线性模式。这对于解决现实世界中复杂的问题非常有用，因为与线性模式相比，数据通常具有更复杂的非线性模式。

提示

让我们通过一个例子来观察一下非线性激活的重要性。首先，回想一下我们在 sigmoid 示例中看到的神经网络的计算。如果我们忽视 b，那将是这样的：

```
h = sigmoid(W*x)
```

假设有一个三层神经网络（W1、W2 和 W3 是层的权重值），其中每层都执行前面的计算；我们可以给出完整计算：

```
h = sigmoid(W3*sigmoid(W2*sigmoid(W1*x)))
```

但是，如果我们删除非线性激活（sigmoid），我们就可以得到：

```
h = (W3 * (W2 * (W1 *x))) = (W3*W2*W1)*x
```

因此，在没有非线性激活的情况下，可以将三个层降为单个线性层。

如果没有各层之间的非线性激活，深度神经网络就将是一堆相互叠加的线性层而已，而且一组线性层基本上可以压缩成一个更大的线性层。综上所述，如果没有非线性激活，我们就无法创建具

有多个层的神经网络。

现在，我们将列出神经网络中两种常用的非线性激活函数以及它们是如何在 TensorFlow 中实现的：

```
#x 的 Sigmoid 激活由 1 / (1 + exp (-x) ) 给出
tf.nn.sigmoid(x,name=None)
#x 的 ReLU 激活由 max(0,x)给出
tf.nn.relu(x, name=None)
```

①卷积运算

卷积运算是一种广泛使用的信号处理技术。对于图像，卷积运算可以给出不同的图像效果。这里，图 3-13 给出了使用卷积进行边缘检测（包括横向边缘检测和纵向边缘检测）的示例。我们可以通过在图像顶部移动卷积过滤器以便在每个位置产生不同的输出来实现边缘检测，如图 3-14 所示（在本书第 5 章介绍卷积神经网络时会详细解读卷积运算的工作原理）。具体来说，在每个位置，我们使用与卷积过滤器重叠的图像块（与卷积过滤器相同的大小）对卷积过滤器中的元素进行逐元素乘法，并获取乘法的总和。

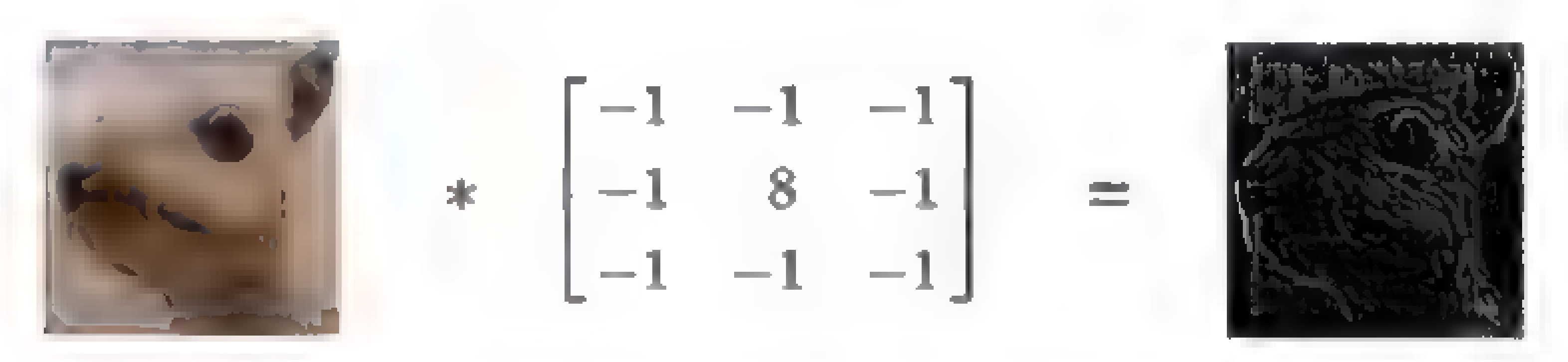


图 3-13　利用卷积运算在图像中进行边缘检测示意图

提示

源自 https://en.wikipedia.org/wiki/Kernel_(image_processing)。

以下是卷积运算的实现：

```
x = tf.constant(
            [[
               [[1],[2],[3],[4]],
               [[4],[3],[2],[1]],
               [[5],[6],[7],[8]],
               [[8],[7],[6],[5]]
         ]],
             dtype=tf.float32)
   x_filter = tf.constant(
       [
         [
           [[0.5]],[[1]]
         ],
         [
```

```
        [[0.5]],[[1]]
      ]
  ],
     dtype=tf.float32)
    x_stride = [1,1,1,1]
    x_padding = 'VALID'
    x_conv = tf.nn.conv2d(
    input=x, filter=x_filter,
    strides=x_stride, padding=x_padding
  )
```

这里，对于 tf.conv2d(...)方法中涉及的 input 、filter 和 stride 等参数格式而言，TensorFlow 对它们的要求是很精确的，下面我们将对这些参数（input、filter、strides、padding）做进一步的解释。

- 输入（input）：通常是四维张量，其尺寸应按[batch_size，height，width，channels]排序。
  - ★ **batch_size**：这是一批数据中的数据量（例如，输入的图像和单词等）。我们通常按批量处理数据，因为模型可以使用大型数据集进行深入学习。在给定的训练步骤（step）中，我们随机抽样一小部分可以大致代表完整的数据集的数据，然后重复足够多次该操作，我们便可以很好地逼近这个完整的数据集。此 batch_size 参数与我们在 TensorFlow 输入管道示例中讨论的参数相同。
  - ★ **height and width**：这是输入的高度和宽度。
  - ★ **channels**：这是输入的深度（例如，对于 RGB 图像，将为 3 通道）。
- 过滤器（filter）：这是一个四维张量，表示卷积运算的卷积窗口。过滤器尺寸应为[height，width，in_channels，out_channels]。
  - ★ height and width：这是滤镜的高度和宽度（通常小于输入的高度和宽度）。
  - ★ in_channels：这是图层输入的通道数。
  - ★ out_channels：这是在图层输出中生成的通道数。
- 步幅（strides）：这是一个包含四个元素的列表，具体为 [batch_stride，height_stride，width_stride，channels_stride]。
- 填充（padding）：这里可以选择['SAME'，'VALID']中的任何一个选项。它能够决定如何处理输入边界附近的卷积运算。VALID 操作是在没有填充的情况下执行卷积。如果我们用大小为 h 的卷积窗口、卷积长度为 n 的输入，这将给出输出的尺寸（或大小）。输出尺寸的减小会严重限制神经网络的深度。SAME 将零填充到边界，使输出具有与输入相同的高度和宽度。

为了更好地了解过滤器大小、步幅和填充是什么，请参见图 3-14（我们在本书第 5 章介绍卷积神经网络时做详细解读）。

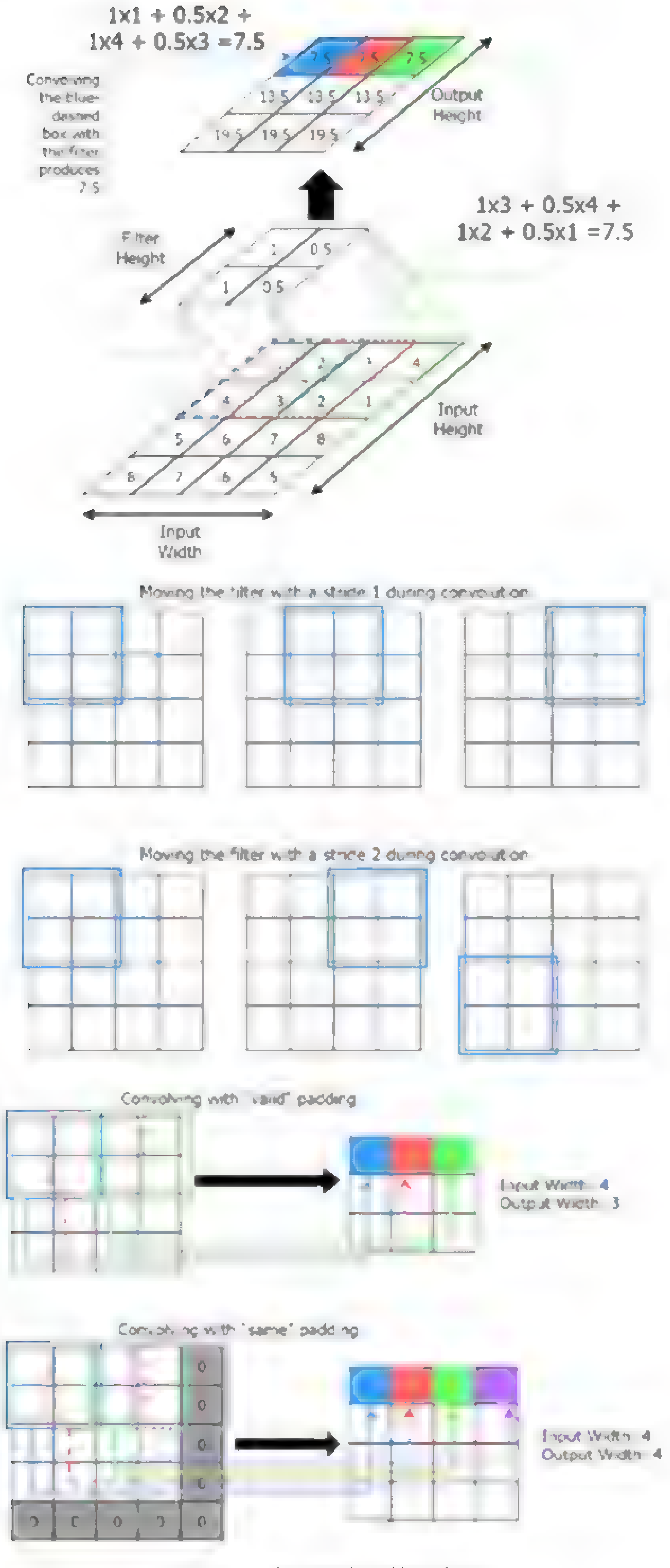


图 3-14 卷积网络运算示意图

②池化操作

池化运算与卷积运算的行为类似，但最终输出是不同的。我们这里选取的是该位置中图像 patch 的最大元素，而不是输出过滤器和图像 patch 中按元素相乘得到的总和（我们在在本书第 5 章介绍卷积神经网络中会做详细解读），如图 3-15 所示。

```
x = tf.constant(
              [[
                 [[1],[2],[3],[4]],
                 [[4],[3],[2],[1]],
                 [[5],[6],[7],[8]],
                 [[8],[7],[6],[5]]
              ]],
             dtype=tf.float32)

        x_ksize = [1,2,2,1]
        x_stride = [1,2,2,1]
        x_padding = 'VALID'
        x_pool = tf.nn.max_pool(
                 value=x, ksize=x_ksize,
                  strides=x_stride, padding=x_padding
          )
```

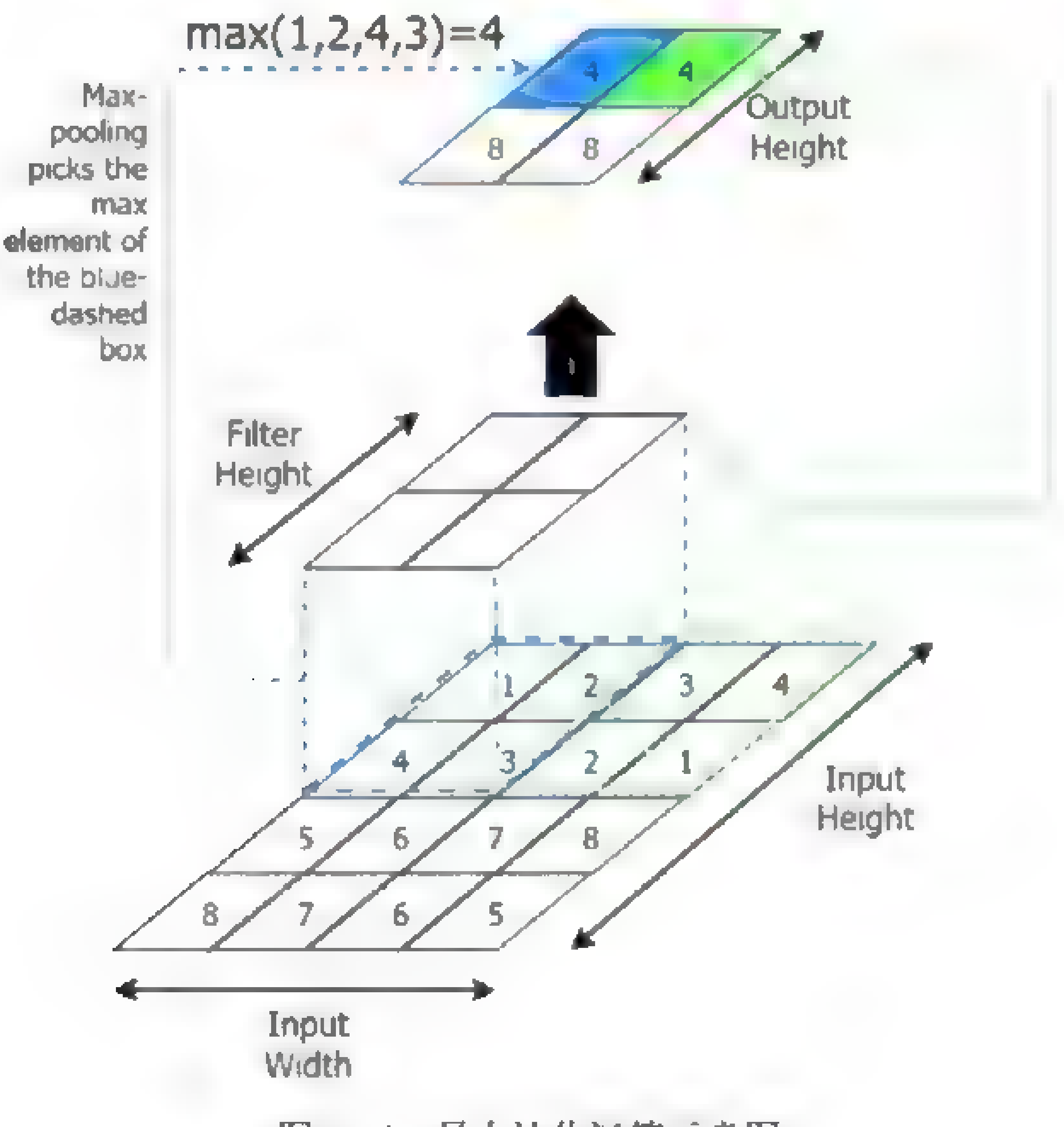


图 3-15　最大池化运算示意图

代码运行后返回的结果如下（完整代码和结果，读者可以查看代码文件中的“ .维操作（2D

卷积和 2D 最大池化）”部分）：

```
[[[[ 4.]
[ 4.]],
[[ 8.]
[ 8.]]]]
```

③定义损失

为了让神经网络模型能够学习到有用的东西，我们需要定义一个损失函数。这里有几种可以自动计算 TensorFlow 中损失的函数。其中，tf.nn.l2_loss 是均方误差损失函数，tf.nn.softmax_cross_entropy_with_logits_v2 是交叉熵损失函数。交叉熵损失函数在分类任务中能够使模型表现更佳。这里涉及均方误差损失函数和交叉熵损失函数的代码如下：

```
x = tf.constant([[2,4],[6,8]],dtype=tf.float32)
x_hat = tf.constant([[1,2],[3,4]],dtype=tf.float32)
# MSE = (1**2 + 2**2 + 3**2 + 4**2)/2 = 15
MSE = tf.nn.l2_loss(x-x_hat)
# 神经网络中用于优化网络的常见损失函数
# 使用 logits（最后一层的归一输出）代替输出来计算交叉熵，会使数值获得的更加稳定
y = tf.constant([[1,0],[0,1]],dtype=tf.float32)
y_hat = tf.constant([[3,1],[2,5]],dtype=tf.float32)
# 此函数并不能平均所有数据点的交叉熵损失，我们需要使用 reduce_mean 函数手动实现这一点
CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=y_hat,
labels=y))
```

④神经网络的优化

在定义了神经网络的损失函数之后，我们的目标是随着时间的推移尽量减少这种损失，这个过程就是常说的模型优化工作。换言之，优化器的目标是找到为所有输入提供最小损失的神经网络参数（权重值和偏差）。TensorFlow 为我们提供了几种不同的优化器，因此我们不需要从头开始实现相关模型。

图 3-16 说明了一个简单的优化问题，并显示了优化是如何随时间变化的。该曲线可以想象为损失曲线（对于高维空间的情况，我们称之为损失面），其中 x 可以被认为是神经网络的参数（在这种情况下，是一个单一权重值的神经网络），y 可以被认为是损失。我们初步估计起始点是 x=2 位置。从这一点开始，我们使用优化器来实现在 x=0 处得到最小的 y（损失）。然而，在实际问题中，损失表面不会像图 3-16 中所示的这么简单，它会更加复杂。

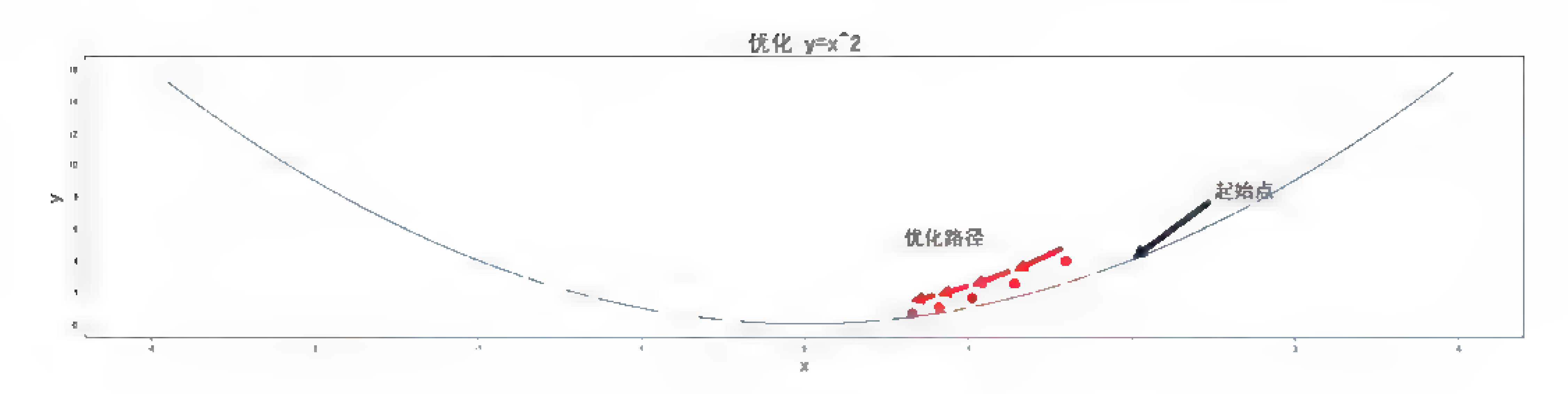


图 3-16　优化过程示意图

在此示例中，我们使用的是常见的梯度下降优化法：GradientDescentOptimizer。learning_rate 参数表示在最小化方向上的步长（图 3-16 中两个圆点之间的距离）：

```
    #优化器起到调整神经网络参数的作用，以便最小化工作任务中的错误
    tf_x = tf.Variable(tf.constant(2.0,dtype=tf.float32),name='x')
    tf_y = tf_x**2
    minimize_op = tf.train.GradientDescentOptimizer(learning_rate=0.1).
minimize(tf_y)
```

完整代码详见代码文件 3_tensorflow_introduction.ipynb 中随机优化（Stochastic Optimization）部分。执行该部分代码后，除了会得到图 3-16 之外，还会得到如下结果：

```
    第 1 个步长上, x: 1.28 , y: 2.5600002
    第 2 个步长上, x: 1.0239999 , y: 1.6384
    第 3 个步长上, x: 0.8191999 , y: 1.0485759
    第 4 个步长上, x: 0.6553599 , y: 0.6710885
```

从上面的代码运行结果来看，显然，当我们每次调用 session.run(minimize_op)执行损失最小化运算时，将会接近 tf_x 值，进而可以得到最小的 tf_y 值。

⑤控制流操作

顾名思义，控制流操作控制图中的执行顺序。例如，假设我们需要按以下顺序执行计算：

```
    x = x+5
    z = x*2
```

实际上，如果 x = 2，我们应该得到 z = 14。下面，让我们尝试以一种简单的方法来实现这一点：

```
    session = tf.InteractiveSession()
    x = tf.Variable(tf.constant(2.0), name='x')
    x_assign_op = tf.assign(x, x+5)
    z = x*2
    tf.global_variables_initializer().run()
    print('z=',session.run(z))
    print('x=',session.run(x))
```

```
session.close()
```

我们期望的输出结果是 x = 7 和 z = 14，而在 TensorFlow 中，上面的代码运行结果却是 x = 2 和 z = 14。引起这种错误的原因是，TensorFlow 不关心对象的执行顺序，除非我们在程序中给出明确的执行顺序。我们本部分讨论的控制流操作，就可以实现执行指定顺序的操作。为了得到期望的运行结果（x = 7 和 z = 14），我们需要对上述代码进行调整，具体如下：

```
session = tf.InteractiveSession()
x = tf.Variable(tf.constant(2.0), name='x')
with tf.control_dependencies([tf.assign(x, x+5)]):
                z = x*2
tf.global_variables_initializer().run()
print('z=',session.run(z))
print('x=',session.run(x))
session.close()
```

这样一来，我们就可以得到想要的结果（x = 7 和 z = 14）了。这里，tf.control_dependencies（...）方法是确保在执行嵌套操作之前将会优先执行参数传递给它的运算操作。读者也可以在代码文件“调用 tf.control_dependencies(...)方法”部分执行这些代码并查看运行结果。

3.10 变量作用域机制

3.10.1 基本原理

目前为止，我们已经对 TensorFlow 架构和 TensorFlow 客户端的实现有了基本的认识。但是，在深度学习过程中，一方面，我们需要减少训练参数的个数（比如 CNN 和 LSTM 模型）或是面对多机多卡并行化训练大数据大模型（比如数据并行化）等情况；另一方面，当我们的深度学习模型变得异常复杂的时候，往往存在大量的变量和调用方法，如何有效地维护这些变量名称和方法名称的唯一性（即不重复），同时又能维护好一个条理清晰的图（Graph）就变得非常重要了。这时，变量共享机制就变得非常重要。比如，我们构建 CNN、LSTM 等模型时，需要使用很多变量集去验证权重值（Weight）和偏差（Bias）等训练参数，非常希望在输入不同的数据时这些参数是可以共享的（本书 5.3.4 小节“参数共享机制”部分会进行详细解读）。过去，我们创建一个全局变量就可以使用了，但在深度学习中则不可以，不方便管理而且使代码的封装性受到极大影响。所以，TensorFlow 提供了一种变量管理方法：变量作用域机制，以此解决上面出现的问题。

关于变量作用域机制，有的文档也叫共享变量机制，根据笔者查阅的文献资料，大部分文章的参考资料来自于 TensorFlow 官方说明（详见 https://tensorflow.google.cn/guide/variables）。TensorFlow 中是通过调用四个函数来进行变量作用域共享的，这四个函数是 tf.Variable(<variable name>)、tf.get variable(<variable name>)、tf.name scope(<scope name>)和 tf.variable scope(<scope name>)。下面，我们具体来看一下这几个函数。

如果使用 Variable，那么每次都会新建变量，但是大多数时候我们是希望一些变量可以重用的，所以就用到了 get variable()。get variable()会去搜索变量名称，搜索结果如果没有就新建变量，如果有就直接使用该变量。既然用到变量名称，这就会涉及名称域的概念。通过不同的域来对变量名称加以区分，因为如果让我们给所有变量都直接取不同名字其实是非常辛苦的且没必要，这就是为什么会用到作用域(Scope)的概念了。name scope 主要用于图(Graph)中的各种运算，variable scope 可以通过设置 reuse 标志以及初始化方式来影响作用域中的变量。当然对我们而言，还有一个更直观的感受就是：在使用 tensorboard 可视化的时候用名字作用域进行封装后会更清晰。

3.10.2　通过示例解读

假设我们需要一个执行某种计算的函数，给定 w，需要计算 x * w + y ** 2。让我们编写一个 TensorFlow 客户端：

```
import tensorflow as tf
session = tf.InteractiveSession()
def very_simple_computation(w):
      x = tf.Variable(tf.constant(5.0, shape=None, dtype=tf.float32),name='x')
      y = tf.Variable(tf.constant(2.0, shape=None, dtype=tf.float32),name='y')
      z = x*w + y**2
      return z
```

这里，为了得到想要的结果，我们可以调用 session.run（very_simple_computation（2））函数（当然是在调用 tf.global_variables_initializer().run()函数之后）。实际上，每次调用这个函数时都会创建两个 TensorFlow 变量。在多次调用该方法(在面向对象程序设计中把封装的函数也称为方法)时，图（Graph）中 x 和 y 变量不会被替换，相反，将会保留这些旧变量，并在图（Graph）中创建新的变量，直到内存不足为止。不管如何，最终的结果是正确的。为了更好地验证这个情况，我们在 for 循环中运行 session.run(very_simple_computation(2))方法，并将变量名称也打印出来，循环 10 次的输出结果如下（完整的代码请在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件“变量作用域机制（Variable Scoping）”部分查看）：

```
   14.0
   ['x:0', 'y:0', 'x_1:0', 'y_1:0', 'x_2:0', 'y_2:0', 'x_3:0', 'y_3:0', 'x_4:0',
'y_4:0', 'x_5:0', 'y_5:0', 'x_6:0', 'y_6:0', 'x_7:0', 'y_7:0', 'x_8:0', 'y_8:0',
'x_9:0', 'y_9:0', 'x_10:0', 'y_10:0']
```

每次运行该函数时都会创建一对变量。如果运行这个函数 100 次，那么计算图中将有 198 个过时变量（99x 变量和 99y 变量）。

作用域允许我们重复使用变量，而不是每次调用函数时都创建一个新的变量。我们现在为上面的例子添加变量可复用性的操作，将代码更改为以下内容：

```
def not_so_simple_computation(w):
```

```
        x = tf.get_variable('x', initializer=tf.constant (5.0,
shape=None,dtype=tf.float32))
        y = tf.get_variable('y', initializer=tf.constant(2.0,
shape=None,dtype=tf.float32))
        z = x*w + y**2
        return z
    def another_not_so_simple_computation(w):
        x = tf.get_variable('x', initializer=tf.constant(5.0,
shape=None,dtype=tf.float32))
        y = tf.get_variable('y', initializer=tf.constant(2.0,
shape=None,dtype=tf.float32))
        z = w*x*y
        return z
    # 这是第一次调用，因此将使用以下名称创建变量
    # x => scopeA/x, y => scopeA/y
    with tf.variable_scope('scopeA'):
        z1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
    #我们重复使用已创建的 scopeA/x 和 scopeA/y
    with tf.variable_scope('scopeA',reuse=True):
         z2 = another_not_so_simple_computation(z1)
    # 由于这是第一次调用，因此将使用以下名称创建变量：x => scopeB/x，y => scopeB/y
    with tf.variable_scope('scopeB'):
         a1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
    #我们重复使用已创建的 scopeB/x 和 scopeB/y

    with tf.variable_scope('scopeB',reuse=True):
         a2 = another_not_so_simple_computation(a1)
    # 假设我们想再次重复使用"scopeA"，因为已经创建了变量，
    #所以我们应该在调用时将"reuse"参数设置为 True
    with tf.variable_scope('scopeA',reuse=True):
         zz1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
         zz2 = another_not_so_simple_computation(z1)
```

在这个例子中，如果执行 session.run（[z1，z2，a1，a2，zz1，zz2]）操作，就应该会看到 z1,z2,a1,a2,zz1,zz2 的值按顺序为 9.0,90.0,9.0，90.0,9.0,90.0 值。现在，如果打印变量，应该只看到四个不同的变量：scopeA / x、scopeA / y、scopeB / x 和 scopeB / y。我们现在可以在循环中多次运行它，而不必担心创建冗余变量和内存不足。

现在你可能想知道，为什么我们不能在代码的开头创建这四个变量并在后面的方法中使用它们。如果这样做，就会破坏代码的封装性，因为这样是在明显地依赖于代码之外的内容。

最后，作用域支持了可复用性，同时也保留了代码的封装性。此外，作用域使代码流更直观，减少了错误的可能性，因为我们通过作用域和名称显式地获取变量，而不是使用 TensorFlow 变量分配的 Python 变量。

3.11 实现一个神经网络

目前，我们已经对 TensorFlow 的架构体系和作用域机制有了大体的认知，接下来，我们将实现一个稍微复杂的模型，那就是完全连接的神经网络模型。这里使用的数据集是在深度学习过程中都经常使用的 MNIST 数据集（详见 http://yann.lecun.com/exdb/mnist/），利用神经网络模型能够实现对数字进行分类。

由于这是我们的第一个神经网络示例，因此，我们将逐步介绍其中的关键部分。如果要了解程序从头到尾的运行情况，读者可以在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件“MNIST 数字识别分类”部分自行查验。

3.11.1 数据准备

首先，我们需要调用 maybe_download(...)函数来下载数据集，并调用 read_mnist(...)函数对其进行预处理。这里，read_mnist(...)函数执行以下两个主要步骤：

- 读取数据集的字节流并将其形成适当的 NUPYP.NDARRY 对象。将图像标准化为零均值和单位方差。

```
def read_mnist(fname_img, fname_lbl):
    print('\nReading files %s and %s'%(fname_img, fname_lbl))
    with gzip.open(fname_img) as fimg:
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
        print(num,rows,cols)
        img = (np.frombuffer(fimg.read(num*rows*cols), dtype=np.uint8).
                   reshape(num, rows * cols)).astype(np.float32)
        print('(Images) Returned a tensor of shape ',img.shape)
        # 对图像进行标准化处理
        img = (img - np.mean(img))/np.std(img)
    with gzip.open(fname_lbl) as flbl:
        #flbl.read (8) 读取最多 8 个字节
        magic, num = struct.unpack(">II", flbl.read(8))
        lbl = np.frombuffer(flbl.read(num), dtype=np.int8)
        print('(Labels) Returned a tensor of shape: %s'%lbl.shape)
        print('Sample labels: ',lbl[:10])
    return img, lbl
```

3.11.2 定义 TensorFlow 计算图

为了定义 TensorFlow 计算图，我们首先为输入图像（tf_input）和其对应标签（tf_lab）定义占位符：

```
# 定义输入和输出
tf_inputs = tf.placeholder(shape=[batch_size, input_size], dtype=tf.float32,
name = 'inputs')
tf_labels = tf.placeholder(shape=[batch_size, num_labels], dtype=tf.float32,
name = 'labels')
```

接下来，编写一个 Python 函数，它将首次创建变量。需要说明的是，我们使用作用域来确保变量的可重用性，并确保变量被正确命名：

```
# 定义 TensorFlow 相关变量
def define_net_parameters():
    with tf.variable_scope('layer1'):
        tf.get_variable(WEIGHTS_STRING,shape=[input_size,500],
                initializer=tf.random_normal_initializer(0,0.02))
        tf.get_variable(BIAS_STRING, shape=[500],
                initializer=tf.random_uniform_initializer(0,0.01))

    with tf.variable_scope('layer2'):
        tf.get_variable(WEIGHTS_STRING,shape=[500,250],
                initializer=tf.random_normal_initializer(0,0.02))
        tf.get_variable(BIAS_STRING, shape=[250],
                initializer=tf.random_uniform_initializer(0,0.01))

    with tf.variable_scope('output'):
        tf.get_variable(WEIGHTS_STRING,shape=[250,10],
                initializer=tf.random_normal_initializer(0,0.02))
        tf.get_variable(BIAS_STRING, shape=[10],
                initializer=tf.random_uniform_initializer(0,0.01))
```

下面，我们将定义神经网络的推理过程。这里需要说明一点，就是与没有使用作用域的变量相比，作用域为函数中的代码提供了非常直观的流程。这里的神经网络有三层，具体如下：

- 具有 ReLU 激活的完全连接层（第 1 层）。
- 具有 ReLU 激活的完全连接层（第 2 层）。
- 完全连接的 softmax 层（输出）。

通过作用域，我们将每个层的变量（权重值和偏差）命名为 layer1 / weight、layer1 / bias、layer2 / weight、layer2 / bias、output / weights 和 output / bias。在下面的代码中，它们都具有相同的名称，但作用域是不同的：

```
#在神经网络中定义根据输入进行逻辑推理的计算过程
#logit 是将 softmax 应用到最终输出之前的评估模型

def inference(x):
```

```
        # calculations for layer 1
        with tf.variable scope('layer1',reuse True):
              w,b = tf.get variable(WEIGHTS STRING), tf.get variable(BIAS STRING)
              tf hl = tf.nn.relu(tf.matmul(x,w) + b, name = 'hidden1')

        # calculations for layer 2
        with tf.variable scope('layer2',reuse=True):
              w,b = tf.get_variable(WEIGHTS_STRING), tf.get_variable(BIAS_STRING)
              tf_h2 = tf.nn.relu(tf.matmul(tf_h1,w) + b, name = 'hidden1')

        # calculations for output layer
        with tf.variable_scope('output',reuse=True):
               w,b = tf.get_variable(WEIGHTS_STRING),
tf.get_variable(BIAS_STRING)
               tf_logits = tf.nn.bias_add(tf.matmul(tf_h2,w), b, name = 'logits')

        return tf_logits
```

现在，我们将定义一个损失函数并将损失进行最小化操作。损失最小化操作是通过在最小化损失方向上对神经网络参数进行微移来开展的。TensorFlow 中提供了多种优化器，我们在这里将使用 MomentumOptimizer，它提供了比 GradientDescentOptimizer 更好的最终精度和收敛性：

```
    # 定义损失函数
    tf_loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=inference(tf_
inputs), labels=tf_labels))

    # 定义损失优化函数
    tf_loss_minimize = tf.train.MomentumOptimizer(momentum=0.9,
    learning_rate=0.01).minimize(tf_loss)
```

最后，我们将定义一个操作来检索给定批量输入所预测的 softmax 概率。这个预测值将反过来用于计算神经网络的准确性：

```
    # 定义预测
    tf_predictions = tf.nn.softmax(inference(tf_inputs))
```

3.11.3 运行神经网络

现在，我们实现了运行神经网络所需要的所有基本操作，可以对它进行检查，看它是否有能力学习对图像中的数字进行正确分类：

```
    for epoch in range(NUM_EPOCHS):
        train_loss = []
```

```
        # 训练阶段
        for step in range(train_inputs.shape[0]//batch_size):
            # Creating one-hot encoded labels with labels
            # One-hot encoding dight 3 for 10-class MNIST data set will result in
            # [0,0,0,1,0,0,0,0,0,0]
            labels_one_hot = np.zeros((batch_size, num_labels),dtype=np.float32)
            labels_one_hot[np.arange(batch_size),train_labels[step*batch_size:
(step+1)*batch_size]] = 1.0

            # 运行优化程序
            loss, _ = session.run([tf_loss,tf_loss_minimize],feed_dict={ tf_inputs:
train_inputs[step*batch_size: (step+1)*batch_size,:], tf_labels:
labels_one_hot} )
            train_loss.append(loss)

         test_accuracy = []
        # 测试阶段
        for step in range(test_inputs.shape[0]//batch_size):
            test_predictions = session.run(tf_predictions,feed_dict={tf_inputs:
test_inputs[step*batch_size: (step+1)*batch_size,:]})
            batch_test_accuracy = accuracy(test_predictions,
    test_labels[step*batch_size:(step+1)*batch_size])
    test_accuracy.append(batch_test_accuracy)

        print('Average train loss for the %d
epoch: %.3f\n'%(epoch+1,np.mean(train_loss)))
        train_loss_over_time.append(np.mean(train_loss))
        print('\tAverage test accuracy for the %d
epoch: %.2f\n'%(epoch+1,np.mean(test_accuracy)*100.0))
        test_accuracy_over_time.append(np.mean(test_accuracy)*100)
```

在这段代码中，accuracy(test_prediction, test_tags)是一个函数，它接受一些预测和标签作为输入，并提供准确性（有多少预测与实际标签匹配）。最终，我们会得到类似于图 3-17 所示的示意图，从运行的结果来看，该模型表现良好，读者也可以自行运行代码进行查验。

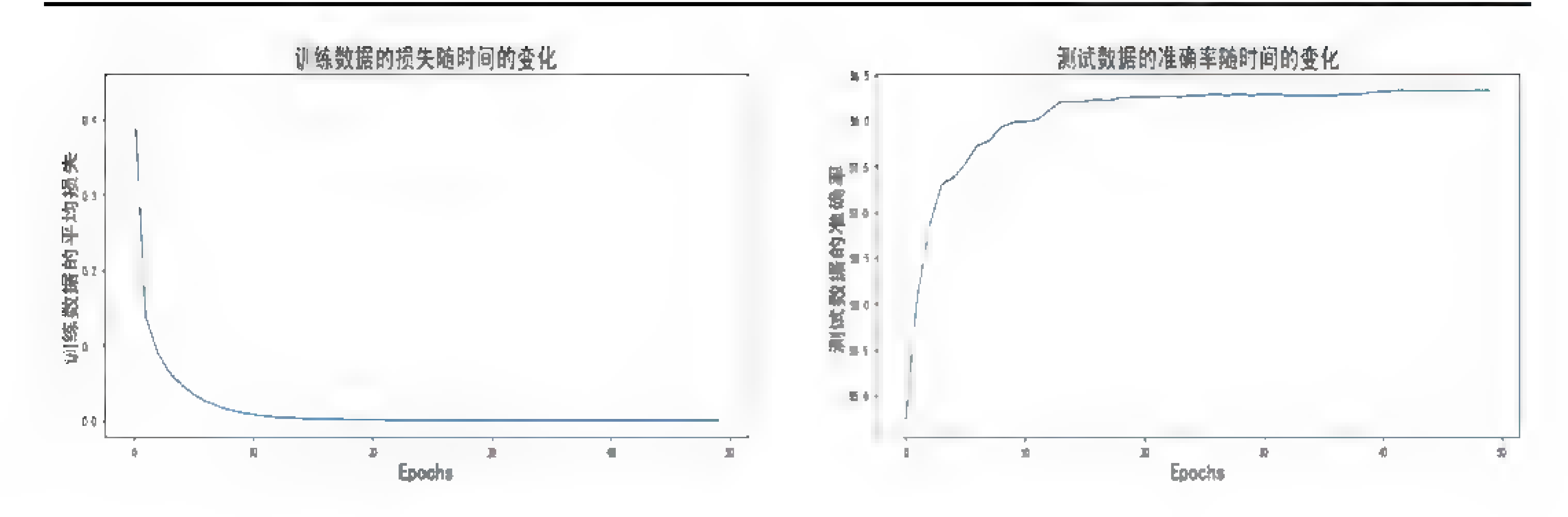


图 3-17 MNIST 数字分类任务的训练损失和测试准确性示意图

3.12 总结

在本章中，我们通过一些示例对算法的主要底层平台（TensorFlow）有了大体上的认知，从而迈出了解决 NLP 任务的第一步。

首先，我们讨论了 TensorFlow 的概念、主要特征及其安装情况。谷歌推出的 TensorFlow 是一个采用数据流图、用于数值计算的开源软件库。它有一些主要特征，例如自动求微分、多语言支持、高度的灵活性、可移植性、性能最优化等。对于 TensorFlow 的安装，我们重点介绍了 Linux 下的安装情况。

其次，我们对于 TensorFlow 的三个主要组成部分（计算图、张量和会话）进行了详细解读。概括起来讲，我们可以用张量表示数据，用计算图搭建神经网络，用会话执行计算图，再优化计算图中线上的权重值（参数）后得到模型。

然后，我们对于 TensorFlow 的工作原理进行了深度解读，了解到 TensorFlow 是一个“client—master—worker”分布式的架构系统。

客户端借助于会话界面可以和master进行交互，把将要触发执行的请求发送给master，而master则会把全部要执行的任务分配给单个或多个 worker，对应的结果通过 master 再返回给客户端。作为专注于执行计算的 worker，任何一个 worker 进程都在管理并使用着整个计算硬件资源，进而采取最优的工作方式来计算子图。

下面我们通过一个 sigmoid 示例对 TensorFlow 进行更深一层的解读，并且在后面结合该示例对 TensorFlow 的客户端做了专门的深度解读。

接下来，我们详细讨论了构成一个典型 TensorFlow 客户端的常见元素：输入、变量、输出和运算（或操作）。输入是我们提供给算法的数据，目的是用于模型的训练和测试。我们讨论了三种不同的输入方式：使用占位符、预加载数据并将数据存储为 TensorFlow 张量以及使用输入管道。然后我们讨论了 TensorFlow 变量，它们与其他张量的区别，以及如何创建和初始化变量。之后，我们讨论了如何使用变量来创建中间和最终的输出。最后，我们讨论了几个可用的 TensorFlow 运算或操作，例如数学运算、矩阵运算、神经网络相关的运算和控制流的操作，这些运算和操作将在

本书后面使用。在对这些常见元素解读的过程中，我们逐步分析并结合代码加以实现，力求让每一位读者都能够更直观地理解其内在逻辑和实现机制。

我们还讨论了在实现 TensorFlow 客户端时如何使用变量作用域来避免某些缺陷。作用域允许我们轻松使用变量，同时也能保持代码的封装性。

最后，我们使用之前学习的所有概念实现了一个神经网络，我们使用三层神经网络对 MNIST 数字数据集进行分类。

下一章，我们将正式开始讲解 NLP 领域的重要模型：词嵌入（Word Embedding）。

第 4 章

词嵌入

根据维基百科，词嵌入（Word Embedding）被定义为自然语言处理（NLP）中的一组语言建模和特征学习技术的集体名称，其中来自词汇表的单词或短语被映射成实数向量。

词嵌入是一种将文本中的词转换为数字向量的方法，我们为了使用标准机器学习算法来对它们进行分析，就需要把这些被转换成数字的向量以数字的形式作为输入。词嵌入过程就是把一个维数为所有词数量的高维空间嵌入到一个维数低得多的连续向量空间中，每个单词或词组被映射为实数域上的向量，词嵌入的结果就是生成了词向量。

我们知道，有一种被称为 One-Hot 编码（被称为独热码或独热编码）的词向量。One-Hot 编码是最基本的向量方法。回顾一下，One-Hot 编码通过词汇表大小的向量表示文本中的词，其中只有对应于该词的项是 1 而所有其他项都是零。

One-Hot 编码的主要问题是无法表示词之间的相似性。在任何给定的语料库中，我们会期望诸如（狗，猫）、（碗,筷子）之类的词对具有一些相似性。使用点积计算向量之间的相似性。点积是向量元素之间元素乘法的总和。在 One-Hot 编码向量的情况下，语料库中任何两个词之间的点积总是为零。

为了克服 One-Hot 编码的局限性，NLP 领域借用了信息检索（IR）技术，使用文档作为上下文来对文本进行矢量化。值得注意的技术是 TF-IDF（https://en.wikipedia.org/wiki/Tf%E2%80%93idf）、潜在语义分析（LSA，https://en.wikipedia.org/wiki/Latent_semantic_analysis）和主题建模（https://en.wikipedia.org/wiki/Topic_model）。然而，这些技术获取到略微不同的、以文档为中心的语义相似性。

词嵌入技术的开发始于 2000 年。词嵌入与以前基于 IR 的技术的不同之处在于，它们使用词作为其上下文，从人类理解的角度来看，这得到了更自然的语义相似性形式。今天，词嵌入是各种 NLP 任务中文本向量化的首选技术，例如词性标注、命名实体识别、文本分类、文档聚类、情感分析、文档生成、问答系统等。

在本章中，我们将对分布式表示、Word2Vec 的两种模型、两种模型对比、Word2Vec 扩展模

型和 GloVe 模型进行探讨和分析，最后我们介绍利用相关模型进行句子分类的案例。这些词嵌入技术已经被证明更有效，且已在深度学习和 NLP 领域中得以广泛采用。

4.1 分布式表示

4.1.1 分布式表示的直观认识

这里有两个句子：

- 北京是中国的首都。
- 华盛顿是美国的首都。

从这两个句子中，我们可以直观地意识到(北京,华盛顿)和(中国，美国)这两对词在某种程度上是相关的，并且相应的词在每对中彼此以相同的方式相关，即：

北京 ： 中国——华盛顿 ： 美国

因此，分布式表示（Distributed Representation）的目的是找到一个变换函数 ϕ，以便将每个词转换为其相关的向量，使得以下形式的关系成立：

ϕ(“北京”)- ϕ(“中国”) ≈ ϕ(“华盛顿”)- ϕ(“美国”)

换句话说，分布式表示旨在将词转换为向量，其中向量之间的相似性与词之间的语义相似性相关。

注意

有的图书上将 GloVe 归入 Word2Vec，这么归类也可以理解，词嵌入实现的技术模型有多种，不论是 Word2Vec 中的两个典型模型（Skip-gram 和 CBOW）还是 GloVe 模型，都是处理词嵌入任务的技术手段，但严格意义上 Word2Vec 和 GloVe 之间还是有些区别的。

4.1.2 分布式表示解读

早期的传统独热编码表示（One-Hot Encoding Representation，如图 4-1 所示），仅仅将词进行符号化，不包含任何语义信息，显然不利于工作任务的有效达成。与独热编码表示技术相对应的就是分布式表示，Harris 于 1954 年提出了分布式假说（Distributional Hypothesis），其观点是上下文中相似的词其语义也相似。这样一来，在该理论指引下，我们可以把信息分布式地存储在向量的各个维度中，这种分布式表示方法具有紧密低维、句法和语义信息易获取的特点。Firth 在 1957 年对分布式假说做了进一步阐述和明确：词的语义由其上下文决定。

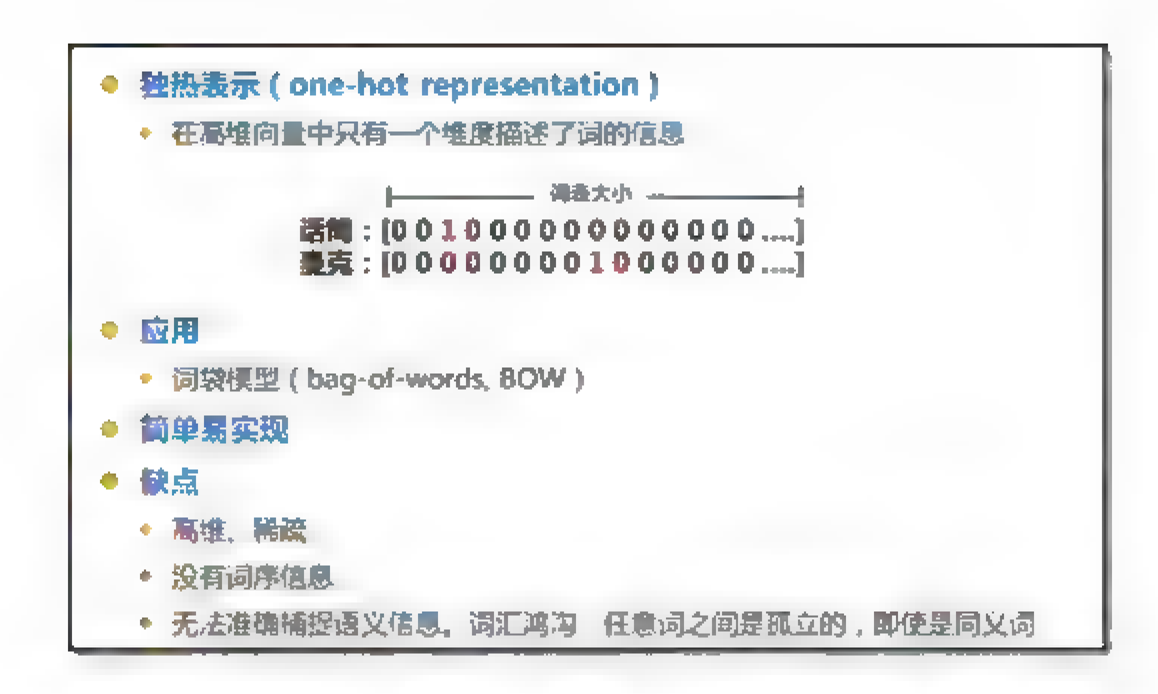


图 4-1　传统的独热编码技术表示

基于分布式假说的词表示方法，根据建模不同，大体可以分为三类：基于矩阵的分布表示、基于聚类的分布表示和基于神经网络的分布表示。尽管这些不同的分布表示方法使用了不同的技术手段获取词表示，但由于这些方法均基于分布式假说，所以它们的核心思想也都由两部分组成：一是选择一种方式描述上下文；二是选择一种模型刻画某个词（“目标词”）与其上下文之间的关系。

下面给出分布式表示方法的三个分类，这里不做过多解读，感兴趣的读者可以仔细研读来斯惟博士的论文《基于神经网络的词和文档语义向量表示方法研究》（中国科学院大学自动化研究所，2016 年 1 月）。

1. 基于矩阵的分布表示

基于矩阵的分布表示主要是构建“词-上下文”矩阵，通过某种技术从该矩阵中获取词的分布表示。矩阵的行表示词，列表示上下文，每个元素表示某个词和上下文共现的次数，这样矩阵的一行就描述了该词的上下文分布情况。常见的上下文有：①文档，即“词-文档”矩阵；②上下文的每个词，即“词-词”矩阵；③n-元词组，即“词-n-元组”矩阵。矩阵中的每个元素为词和上下文共现的次数，通常会利用 TF-IDF、取对数等方法进行加权和平滑。另外，矩阵的维度较高并且非常稀疏的话，可以通过 SVD、NMF 等手段进行分解降维，变为低维稠密矩阵。关于矩阵的分布表示见图 4-2。

● **构建一“词-上下文”矩阵，从矩阵中获取词的表示。矩阵中，每行对应一个词，每列表示一种不同的上下文，矩阵中的每个元素对应相关词和上下文的共现信息。**	
● **构建步骤**	● 选取上下文：词-文档，词-词，词-n 元词组 ● 确定矩阵在中元素的值：共现次数，TF-IDF ● 矩阵分解：SVD,PCA
● **代表模型**	● LSA ● GloVe(Global Vector)

图 4-2　基于矩阵的分布表示

2. 基于聚类的分布表示

通过聚类手段构建词与其上下文之间的关系，代表模型为布朗聚类（Brown Clustering）。

3. 基于神经网络的分布表示

（1）神经网络语言模型（Neural Network Language Model，NNLM）

Xu 等人在 2000 年首次尝试使用神经网络求解二元语言模型。2001 年，Bengio 等人正式提出神经网络语言模型。该模型在学习语言模型的同时也得到了词向量，具体如图 4-3 所示。

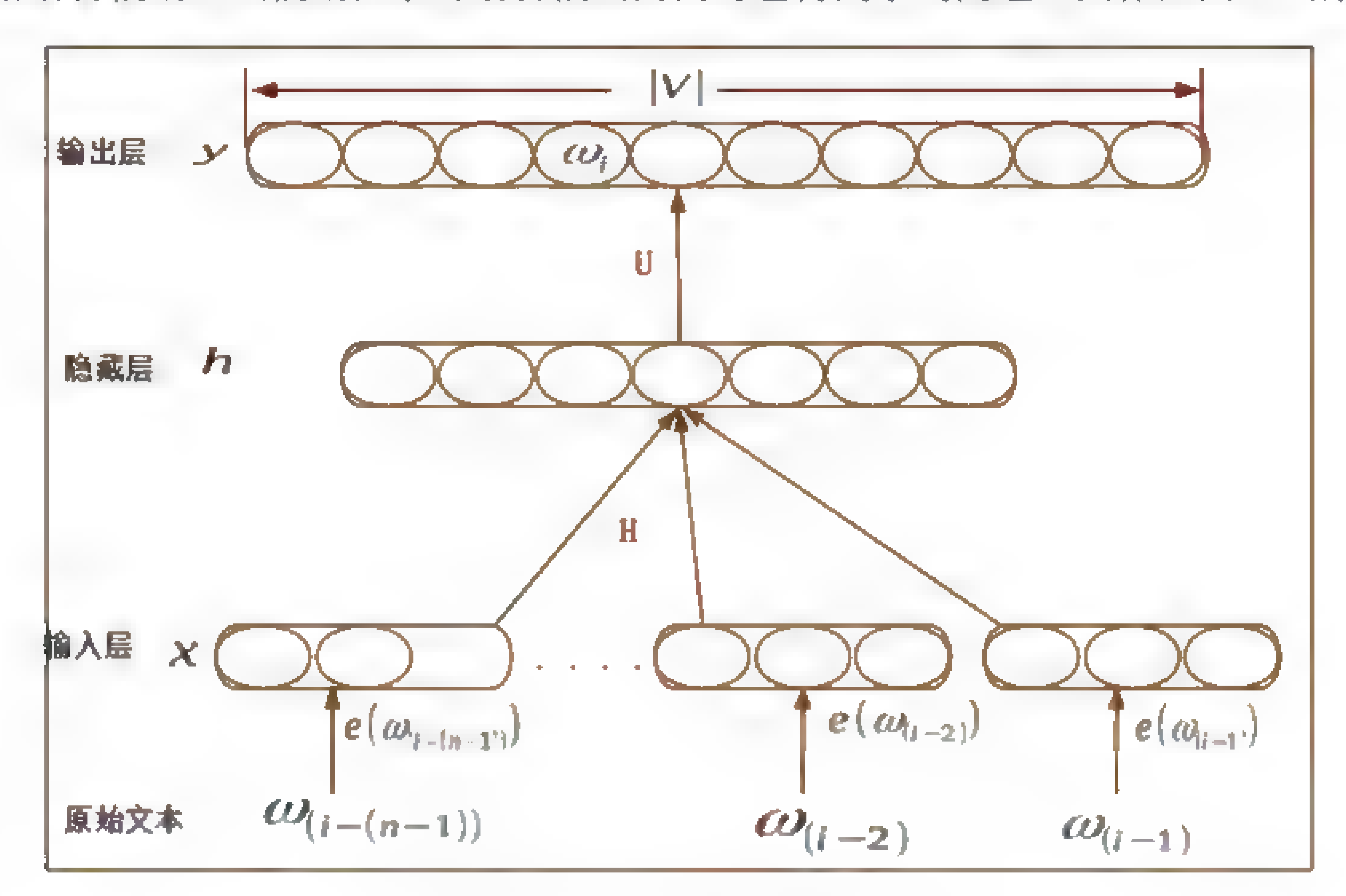


图 4-3 神经网络语言模型（NNLM）

其中，输入层的词向量顺序拼接为$x = \left[e\left(\omega_{(i-(n-1))}\right);\ldots;e\left(\omega_{(i-2)}\right);e\left(\omega_{(i-1)}\right)\right]$；隐藏层 h 和输出层 y 分别为：$h = tanh(b^1 + Hx)$，$y = b^2 + Wx + Uh$；将 y 转成对应的概率值：

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\exp\left(\boldsymbol{y}(w_i)\right)}{\sum_{k=1}^{|V|} \exp\left(\boldsymbol{y}(v_k)\right)}$$

而对于整个语料而言,语言模型需要做以下最大化运算：

$$\sum_{w_{i-(n-1):i} \in \mathbb{D}} \log P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

（2）Log 双线性语言模型（LBL）

2007 年，Mnih 和 Hinton 在神经网络语言模型的基础上提出了 log 双线性语言模型（Log-Bilinear Language Model，LBL），如图 4-4 所示。

NNLM

$$y(\omega_i)=b^2+e'(\omega_i)^T\tanh(b^1+H[e(\omega_{i-n+1});\ldots;e(\omega_{i-2});e(\omega_{i-1})])$$

$$=E(\omega_i;w_{(i-(n-1):(i-1))})$$

LBL

$$E(\omega_i;w_{(i-(n-1):(i-1))})=b^2+e(\omega_i)^T b^1+e(\omega_i)^T H[e(\omega_{(i-(n-1))});\ldots;e(\omega_{(i-2)});e(\omega_{(i-1)})]$$

图 4-4　Log 双线性语言模型

（3）循环神经网络语言模型（RNNLM）

Mikolov 等人提出的循环神经网络语言模型（Recurrent Neural Network based Language Model，RNNLM）直接对 P() 进行建模，RNNLM 可以利用所有的上文信息预测下一个词，其模型结构如图 4-5 所示。

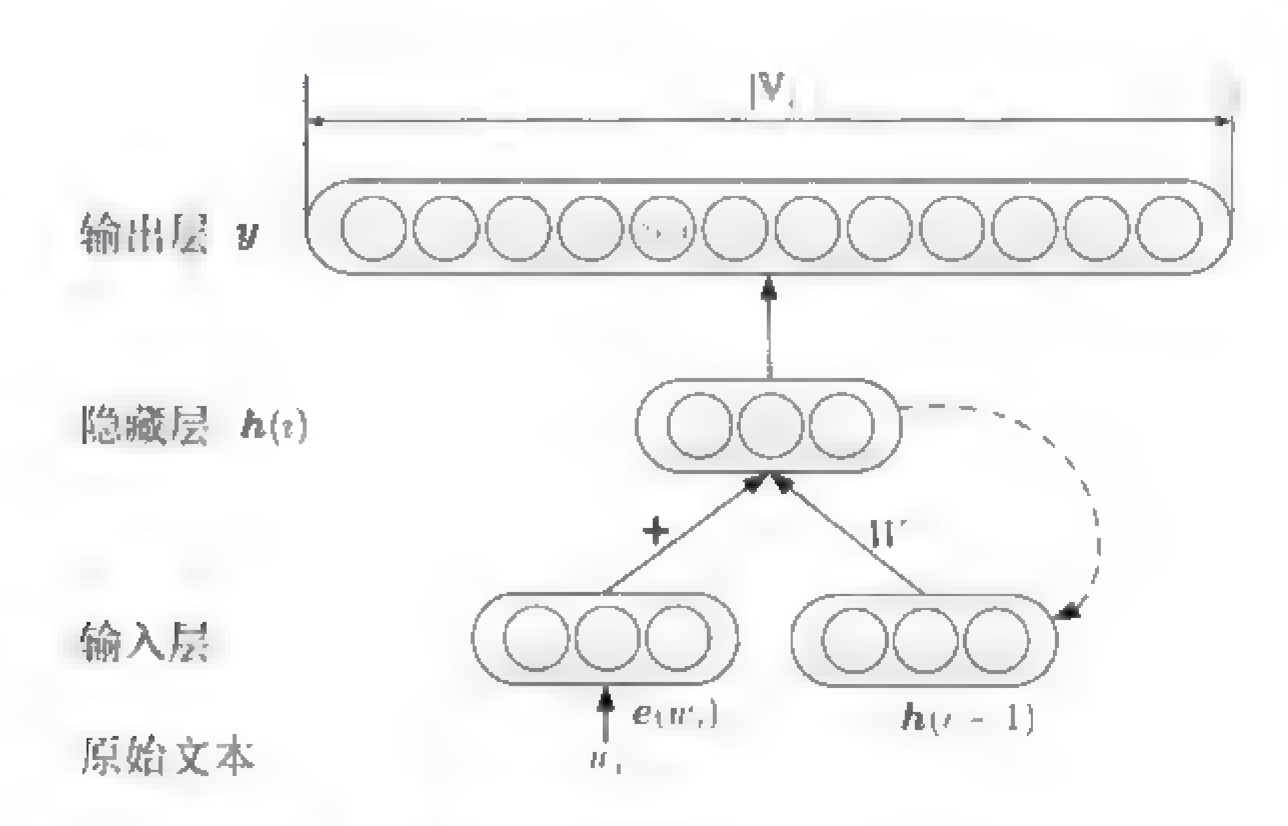


图 4-5　循环神经网络语言模型（RNNLM）

（4）C&W 模型

与前面的三个基于语言模型的词向量生成方法不同，Collobert 和 Weston 在 2008 年提出的 C&W 模型（见图 4-6），是第一个直接以生成词向量为目标的模型。

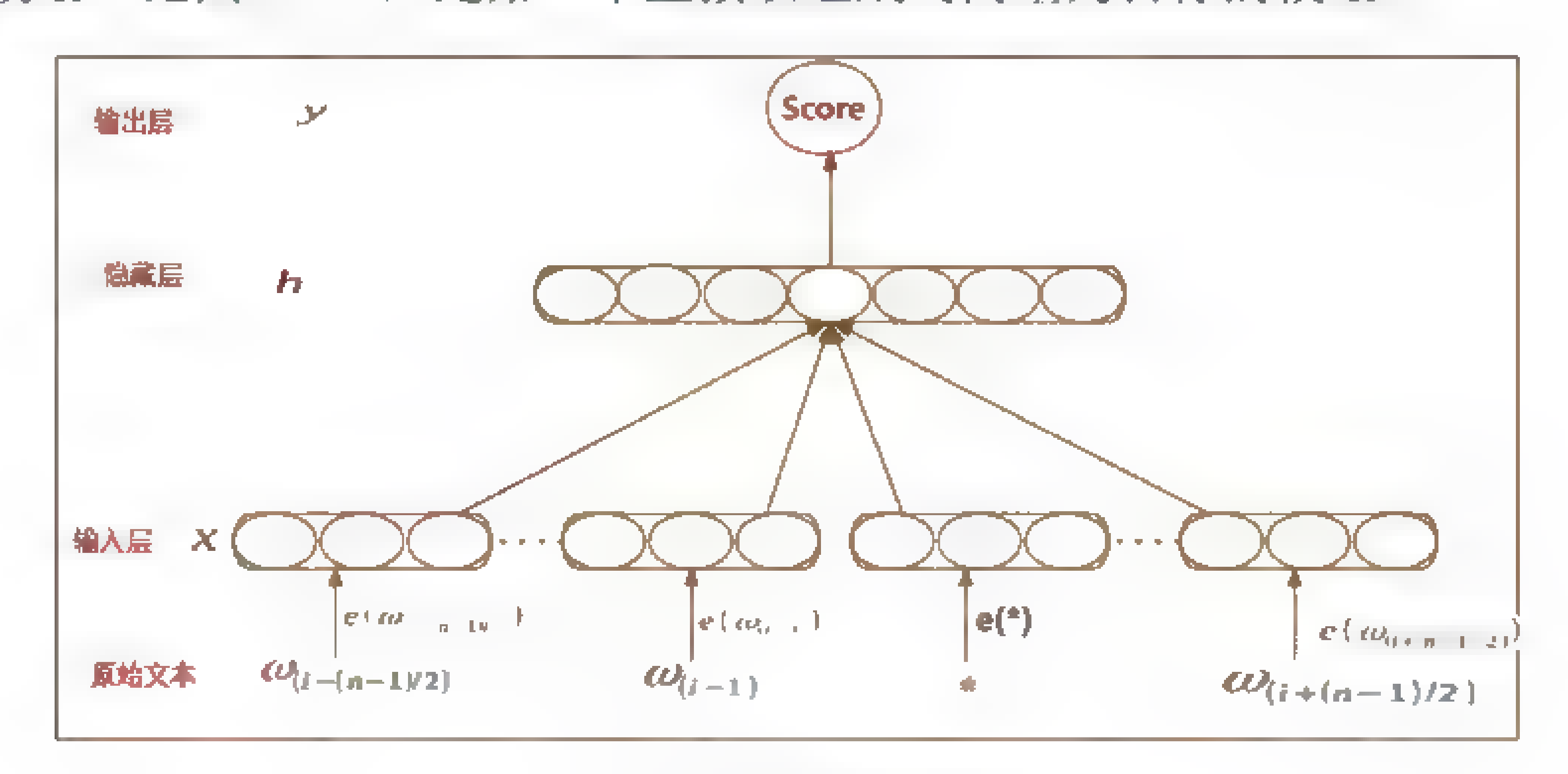


图 4-6　C&W 模型

就 C&W 模型（见图 4-6）与 NNLM（见图 4-3）之间的差异而言，主要在于 C&W 模型将目标词置于输入层，且同时输出层也从语言模型的 |V| 个节点变为一个节点，该节点的数值表示对这组 n 元短语的评分。评分也仅有高低之分，而不存在概率的特性，所以，我们无需进行复杂的归一化操作。C&W 模型使用这种方式把 NNLM 模型在最后一层的 |V| × |h| 次运算降为 |h| 次运算，这样就极大地降低了模型的时间复杂度。这个区别使得 C&W 模型成为神经网络词向量模型中最为特殊的一个，其它模型的目标词均在输出层，只有 C&W 模型的目标词在输入层。

（5）著名的 Word2vec

Word2vec 包含两个典型模型： Skip-gram（SG）和 Continuous Bag-of-Words（CBOW），如图 4-7 所示。

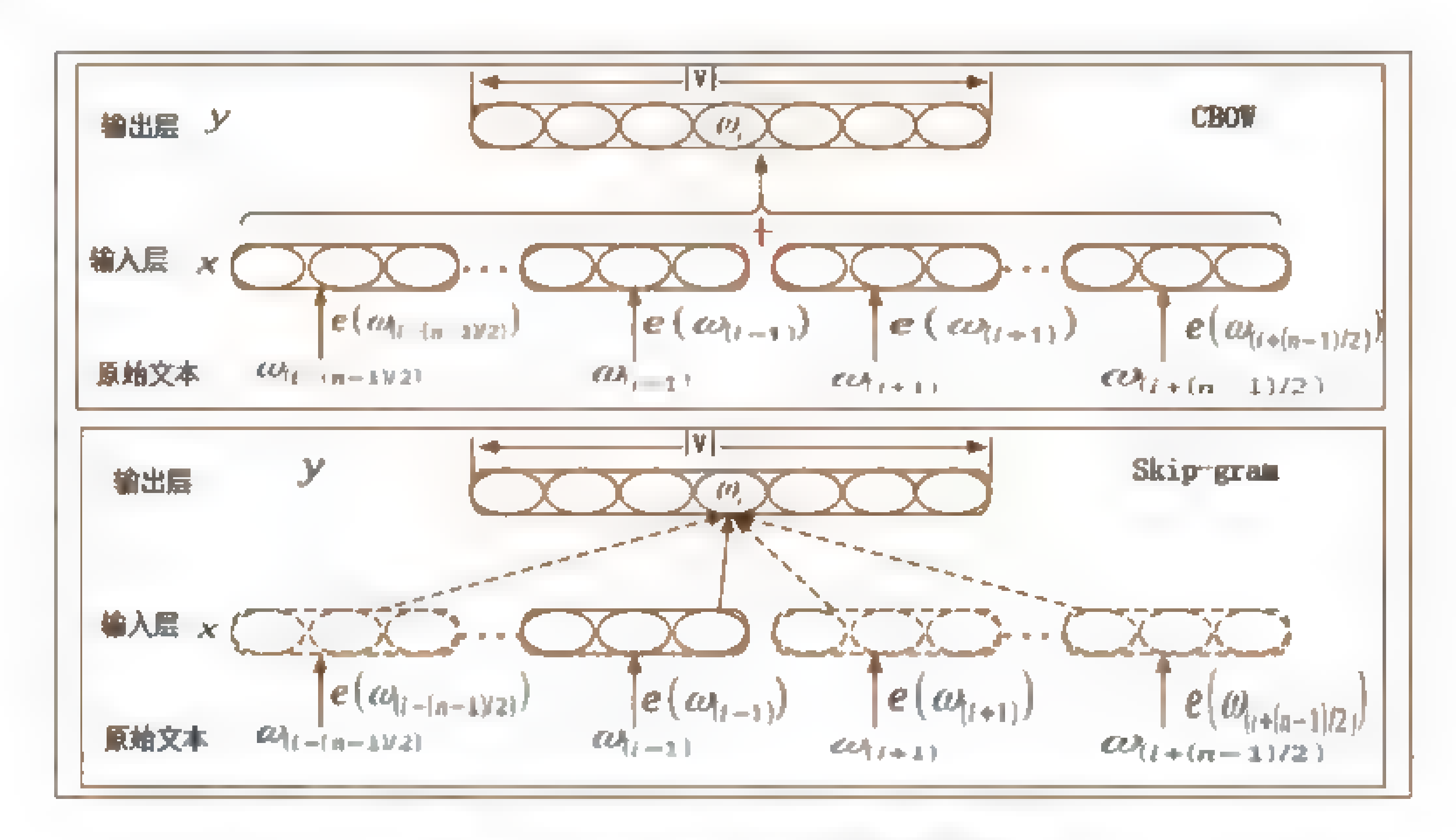


图 4-7 Word2vec 工具中的两个模型

Word2vec 是谷歌在 2013 年推出的一个 NLP 工具，是从大量文本语料中以无监督的方式学习语义知识的一种模型，它的特点是将所有的词向量化，这样我们就能以定量的方式去度量词与词之间的关系，挖掘词之间的联系，所以 Word2Vec 被大量用于自然语言处理（NLP）中。Word2Vec 模型分为两个部分，第一部分是建立模型，第二部分是通过模型获取词向量。Word2Vec 整个建模过程实际上与自编码器（Auto-Encoder）的思想很相似，即先基于训练数据构建一个神经网络，当这个模型训练好以后，我们先不会用这个训练好的模型处理新的任务，而真正需要的是这个模型通过训练数据所学得的参数，其实这与部分传统机器学习模型类似（比如，笔者博客《机器学习实践之预测数值型数据——回归》一文中提到的通过训练算法寻找回归系数），例如隐藏层的权重值矩阵，后面我们将会看到这些权重值在 Word2Vec 中实际上就是我们试图去学习的词向量（Word Vector）。

对于整个语料库而言，CBOW 优化的最大目标为：

$$\sum_{(w,c)\in \mathbb{D}} \log P(w|c)$$

其中，$P(\omega|c)$是 CBOW 模型根据上下文的表示来实现对目标词直接进行的预测，具体如下：

$$P(w|c) \quad \frac{\exp\left(e'(w)^{\mathrm{T}}x\right)}{\sum_{w'\in \mathrm{V}}\exp\left(e'(w')^{\mathrm{T}}x\right)}$$

同样，对于整个语料库而言，Skip-gram 模型通过上下文预测目标词，优化的最大化目标为：

$$\sum_{(w,c)\in \mathbb{D}}\sum_{w_j\in c}\log P(w|w_j)$$

其中，$P(\omega|\omega_j)$为：

$$P(w|w_j) = \frac{\exp\left(e'(w)^{\mathrm{T}}e(w_j)\right)}{\sum_{w'\in \mathrm{V}}\exp\left(e'(w')^{\mathrm{T}}e(w_j)\right)}$$

（6）Order 模型

上面提到的 CBOW 模型和 Skip-gram 模型为了获得更高的性能，在神经网络语言模型或者 log 双线性语言模型的基础上去掉了隐藏层和词序信息。为了更好地分析词序信息对词向量性能的影响，这里有一个新模型，名为"Order"（见图 4-8），意为保留了词序信息。该模型在保留词序信息的同时，去除了隐藏层。

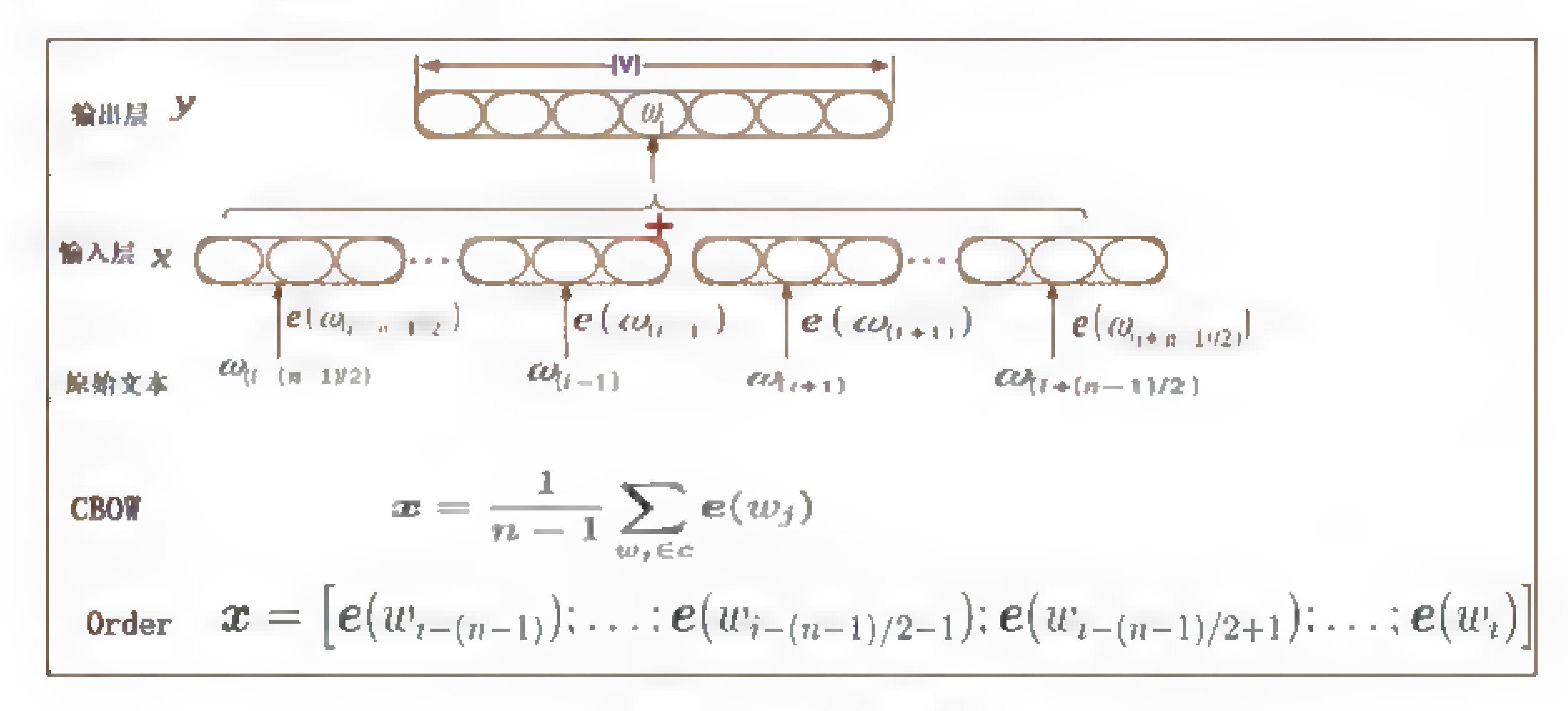


图 4-8　Order 模型

伊万•布罗德罗夫（Ivan Vendrov）、瑞安基洛斯（Ryan Kiros）、Sanja Fidler、拉奎尔•乌塔森（Raquel Urtasun）发表在 ICLR2016 上的论文《*ORDER -E MBEDDINGS OF IMAGES AND LANGUAGE*》，提出了一种新的基于偏序关系的分布式表示向量构建方法，可以用于图像与文本等领域，并在上下位关系预测（Hypernym Prediction）、标题-图像检索（Caption-ImageRetrieval）和自然语言推理（Textual Entailment / Natural Language Inference）等三个任务上进行了实验，取得了不错的效果。文章的最大亮点在于基于偏序关系的向量（Order-Embedding）的构建，如果有时间，建议读者看看原文。

论文提出了一种全新的基于偏序关系的向量学习方法，将维持具有层次关系的向量之间的偏序关系作为学习的目标。上下位关系预测、标题-图像检索和自然语言推理等任务，本质上都是学习图像和文字上偏序关系的实例。如图 4-9 所示，图像标题即为图像的一种抽象表达，而标题本身的表达也抽象形成一个层次结构，这种层次结构即为一种偏序关系。上下位关系预测和自然语言推理

等任务也自然能够转化为一个学习偏序关系的任务。

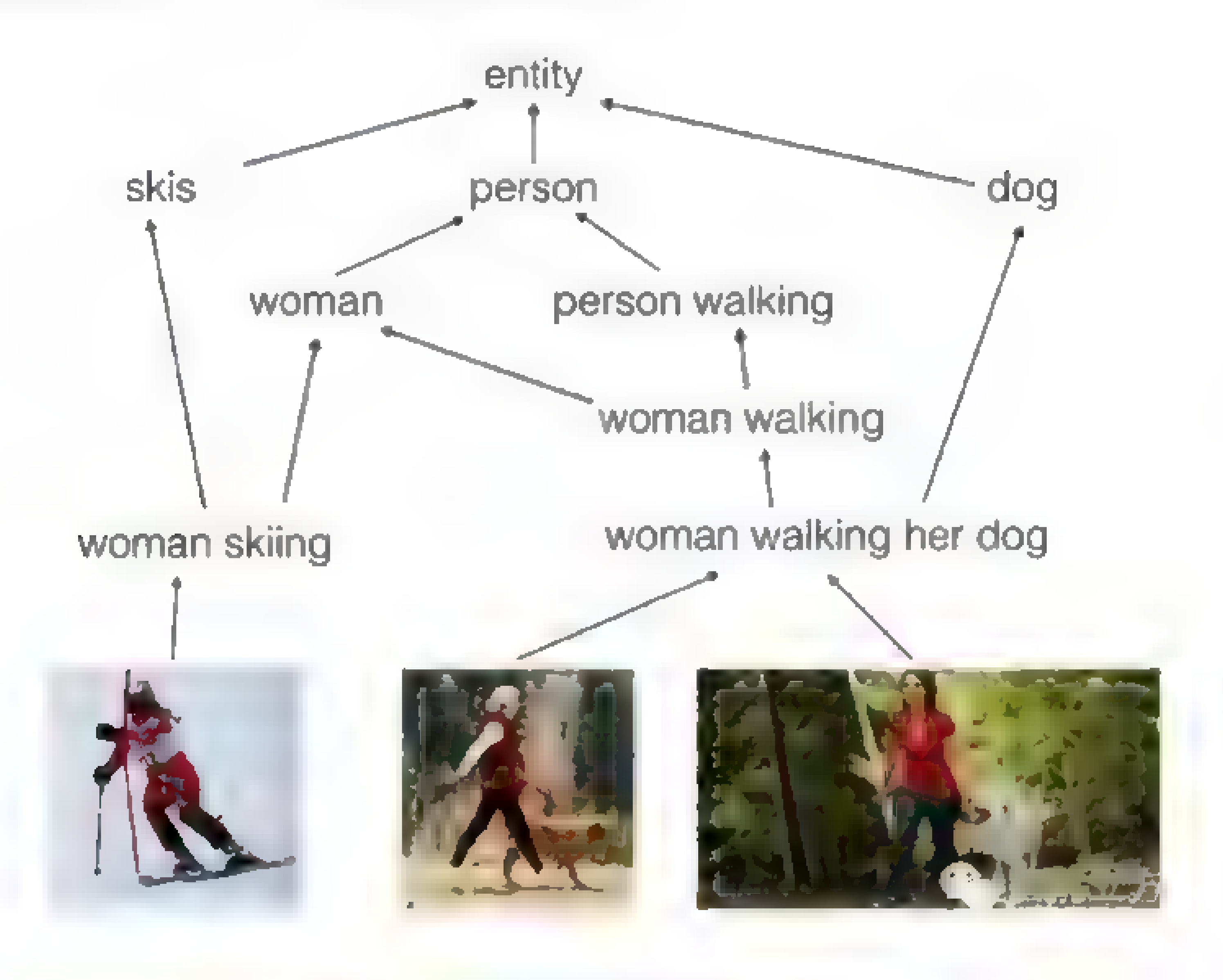


图 4-9 视觉语义层次结构（部分）

4.2 Word2vec 模型（以 Skip-Gram 为例）

Mikolov 等人在 2013 年的文献中，同时提出了 Skip-Gram 和 CBOW（Continuous Bagof-Words）模型，设计两个模型的主要目的是希望用更高效的算法获取词向量。因此，根据前人在 NNLM、RNNLM 和 C&W 模型上的积累，他们简化了当时已有模型且保留了核心部分，便有了这两个模型。常见的 CBOW 和 Skip-Gram 模型结构图如图 4-10 所示。

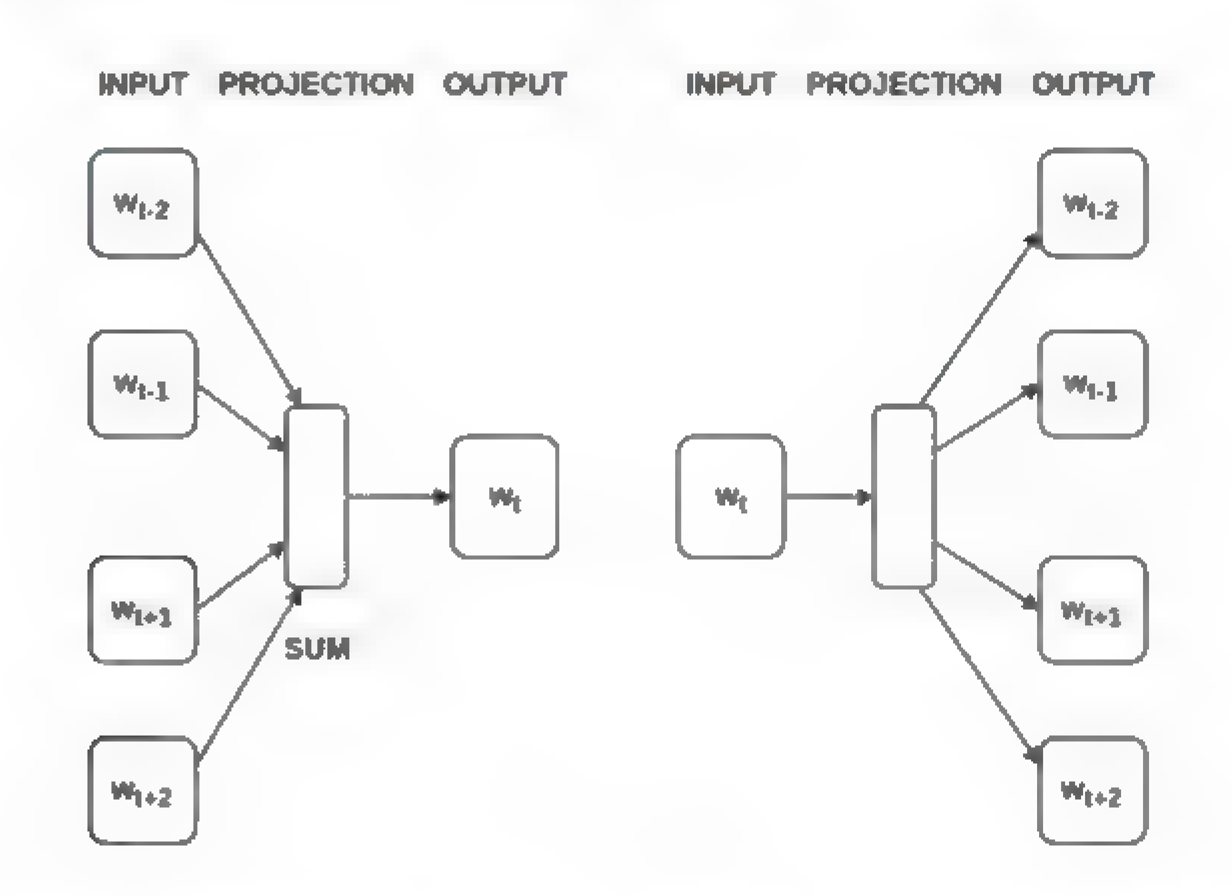


图 4-10 模型结构图：左侧 CBOW 模型，右侧 Skip-Gram 模型

Skip-Gram 算法模型是一种利用文字的上下文来学习词嵌入的算法。让我们在后文一步一步地去理解 Skip-Gram 算法。因为原始的 Skip-Gram 模型没有中间隐藏层，而目前使用的优化 Skip-Gram 模型存在中间隐藏层，为了更好地解读 Skip-Gram 模型，所以这里先讨论优化 Skip-Gram 模型，后面会对原始 Skip-Gram 和优化 Skip-Gram 做一个简单对比分析。

4.2.1　直观认识 Word2vec

上面提到的词向量（Word2vec）是词的数字表示，保留了词之间的语义关系。例如，单词 cat 的向量与单词 dog 的向量就非常相似，而单词 pencil 的向量却与词 cat 的向量差别很大，甚至完全不同。其实，这种词向量间的相似性取决于对应的频率，所以这里讨论一下两个词（如[cat,dog]或者[pencil,cat]）在相同上下文中的使用情况。考虑以下句子：



很显然，在组成上面句子的过程中，无论是 cat 还是 dog，都比 pencil 更加适合填充到空白处。在上面的句子中，pencil 无论从拼写还是语法方面都没有问题，为什么它就是不对呢?是因为这里的 pencil 单词会使得上下文语义产生错误。所以，Word2vec 算法模型使用词的上下文来学习词的数字表示，以便在相同上下文中所使用的词具有相似的词向量。

4.2.2　定义任务

我们现在解读一下 Word2vec 算法的机制情况，后面会继续讲解模型细节，以便我们知道如何实现 Word2vec 算法。为了以无监督方式学习词向量（数据没有被标记），我们需要定义且完成一些任务，具体任务有：

（1）创建格式为[输入词,输出词]的数据元组，其中每个词都表示为一个 One-Hot 向量，均来自原始文本。

（2）定义出一个模型，将 One-Hot 向量作为输入和输出进行训练。

（3）定义一个损失函数，用于预测正确的词（这些词来自于输入词的上下文），以便优化模型。

（4）通过相似词具有相似的词向量来评估模型。

这个流程看似简单，其实在学习词向量的表现上是超强的，下面对相关细节进行解读。

4.2.3　从原始文本创建结构化数据

这部分对于原始文本的操作并不复杂，只是将其置于某个结构中。下面给出一个例子，这里有下面一句话：

```
The cat pushed the glass off the table.
```

由上面这句话，我们创建的数据结构如图 4-11 所示。句子下面的每一行代表一个数据点。蓝色框表示 One-Hot 输入词（中间词，也叫目标词），红色框表示 One-Hot 输出词（上下文窗口中除中间词之外的任何词均称为上下文词）。上下文窗口大小越大，模型的性能就越好。随着数据量的增加，窗口的大小也随之增大，进一步导致计算成本快速上升。注意一点，不要将目标词与神经网络的目标（正确输出）混淆，这是两个完全不同的东西。本节内容暂以 Skip-Gram 模型为例进行解释说明。

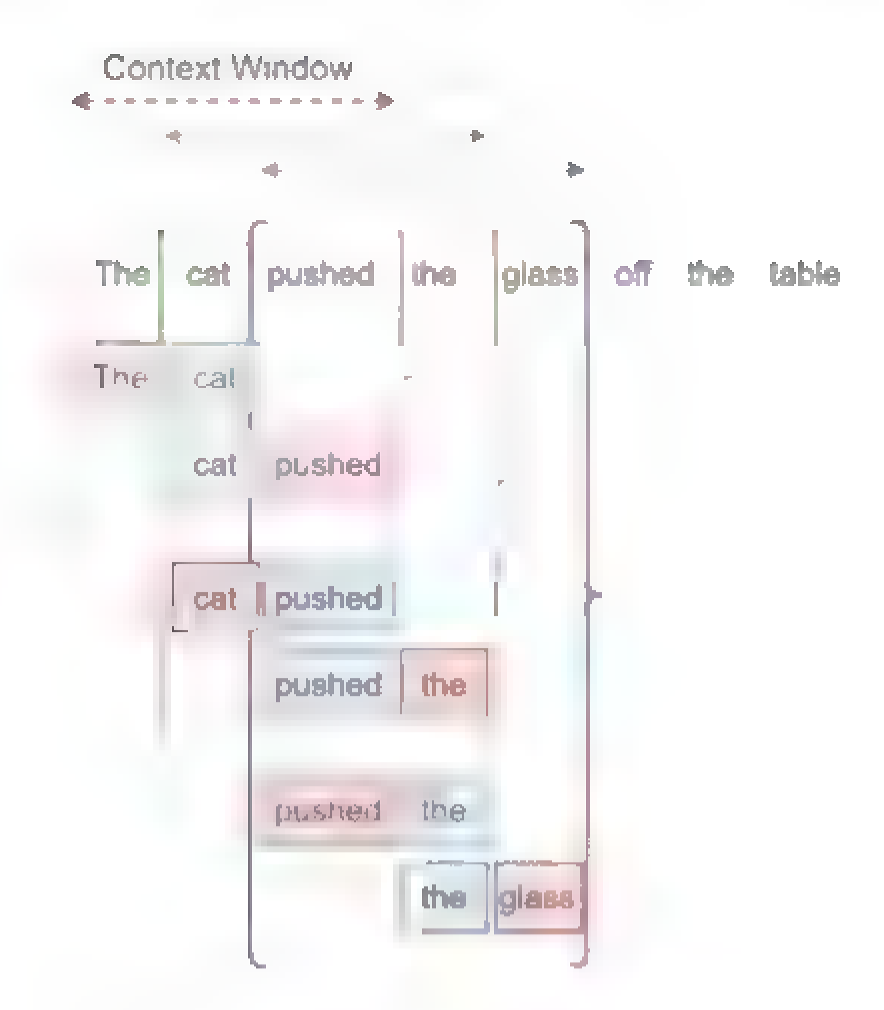


图 4-11 例句数据结构示意图

这里，单个上下文的窗口是指从当前输入词（Input Word，即目标词）的一侧（左边或右边）选取词的数量。

如果我们设置窗口大小 window_size=2，那么我们最终获得窗口宽度为 5，窗口内容就是“'The', 'cat'，'pushed', 'the','glass'”。

注意

网上有专栏文章多处指出，整个窗口大小是 2*window_size = 4，笔者为此查阅多篇国外原文资料，最终认为整个窗口宽度或大小应该为 span=2*window_size+1，有疑惑的地方，建议读者多看几篇国外原文，因为也有个别国外文章中提到 span = 2*window_size，误导了读者。

4.2.4 定义词嵌入层和神经网络

1. 词嵌入层（Embedding Layer，也称为嵌入层）：存储所有词向量

词嵌入层存储词汇表中找到的所有词的词向量。你可以想象到这将是一个巨大的矩阵（[vocabulary size × embedding size]，即[词汇大小×词向量大小]）。这里读者可以自行调整词向

量大小（Embedding Size）。当然，词向量大小越大，模型表现越佳。结合上面的例句，其流程图如图 4-12 所示。

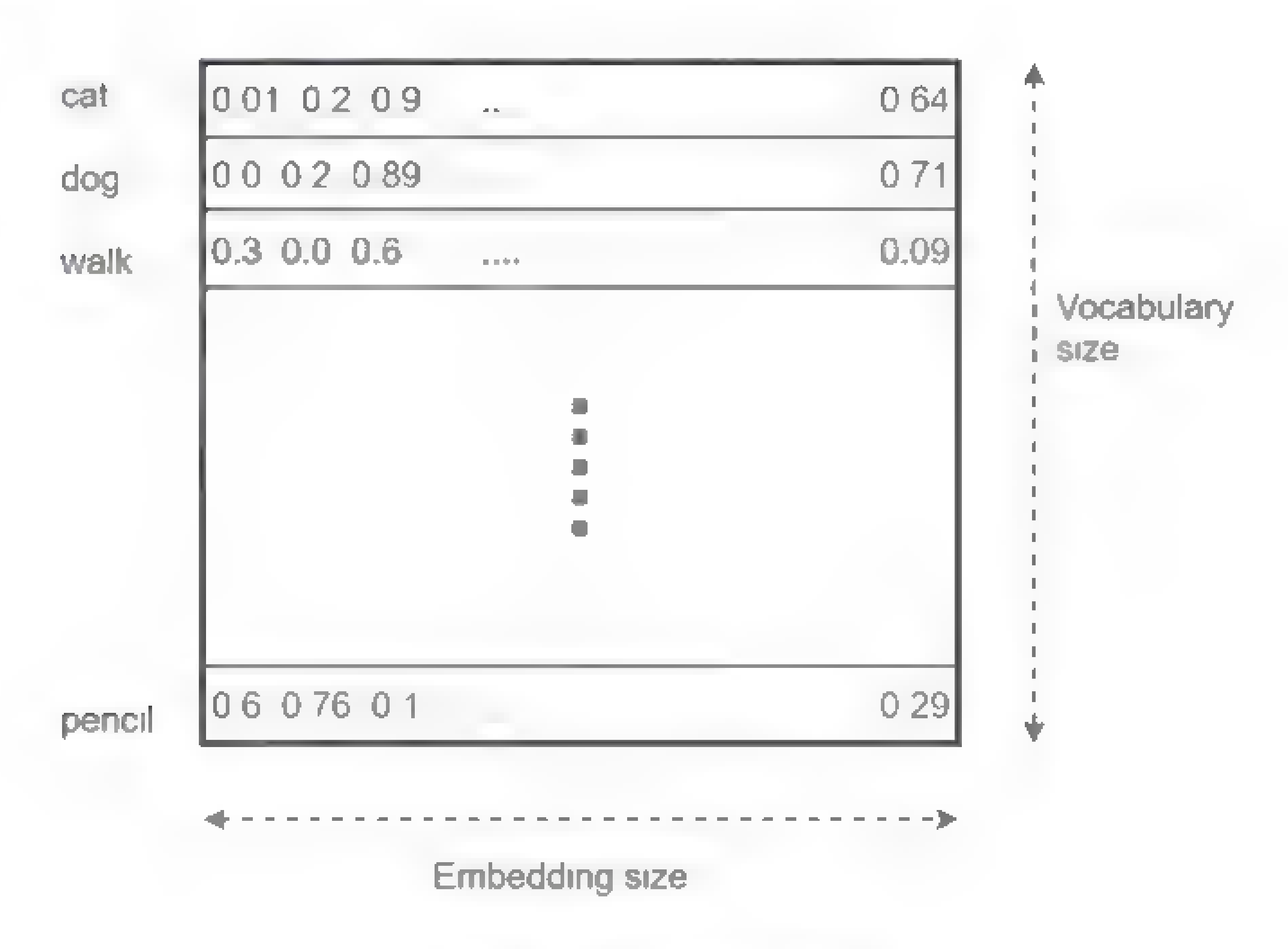


图 4-12　词向量示例图

2. 神经网络：将词向量映射到输出

在训练期间，神经网络利用输入词去尝试预测输出词。然后，我们会惩罚模型的错误分类以及奖励模型的正确分类。这里对模型会话做了一些限制：一次处理单个输入和单个输出。下面是训练期间的流程：

（1）对于给定的输入词（目标词），从词嵌入层中找到相应的词向量。

（2）将词向量输入神经网络，然后尝试预测正确的输出词（上下文词）。

（3）通过比较预测和真实的上下文词，计算损失。

（4）利用损失函数和随机优化器来优化神经网络和词嵌入层。

需要注意的一点是，在计算预测时，我们使用 softmax 激活函数将预测标准化为有效的概率分布。

4.2.5　整合

下面我们可以将所有部分放在一起来看模型的全貌，如图 4-13 所示。

这种数据的局部排列和模型布局是 Word2vec 的 Skip-Gram 算法，也是我们本节要关注的内容。另一种算法称为连续词袋（CBOW）模型，后面也会单独阐述。

至此，我们可以对于 Skip-Gram 模型的概念结构和实施结构做个小的总结。图 4-14 所示为概念结构图，图 4-15 所示为实施结构图。

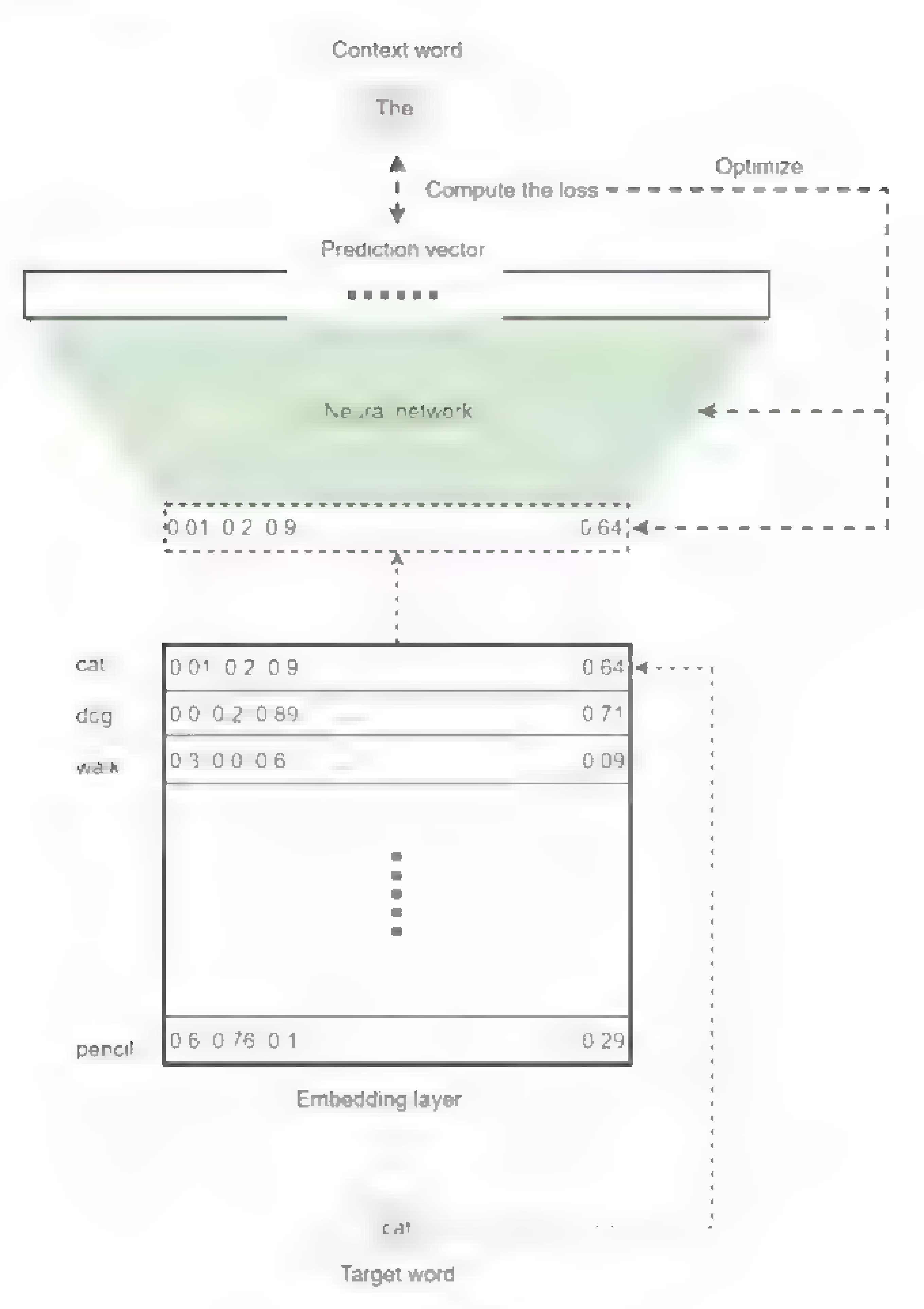


图 4-13 模型全貌示例

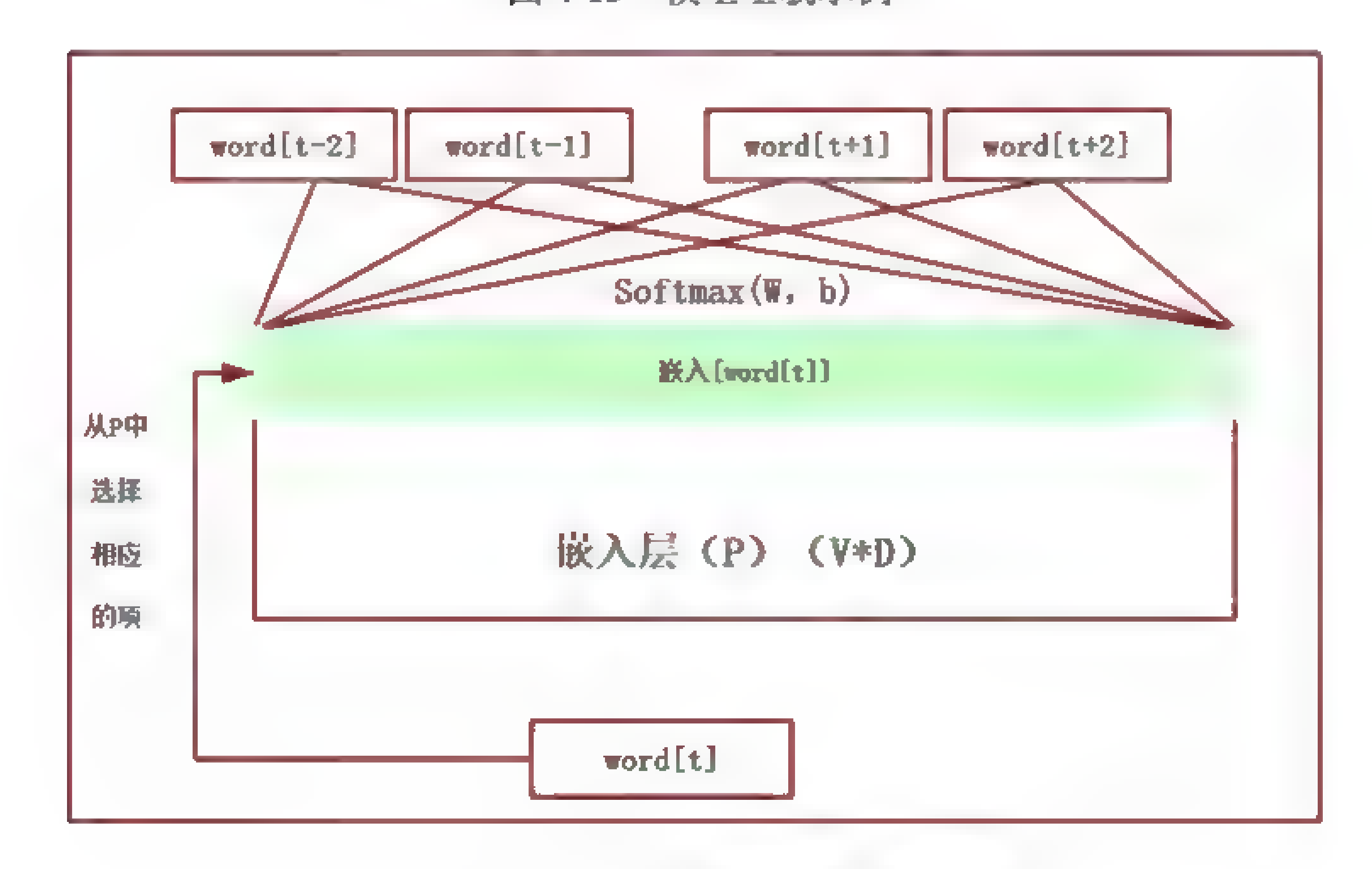


图 4-14 Skip-Gram 模型概念架构图

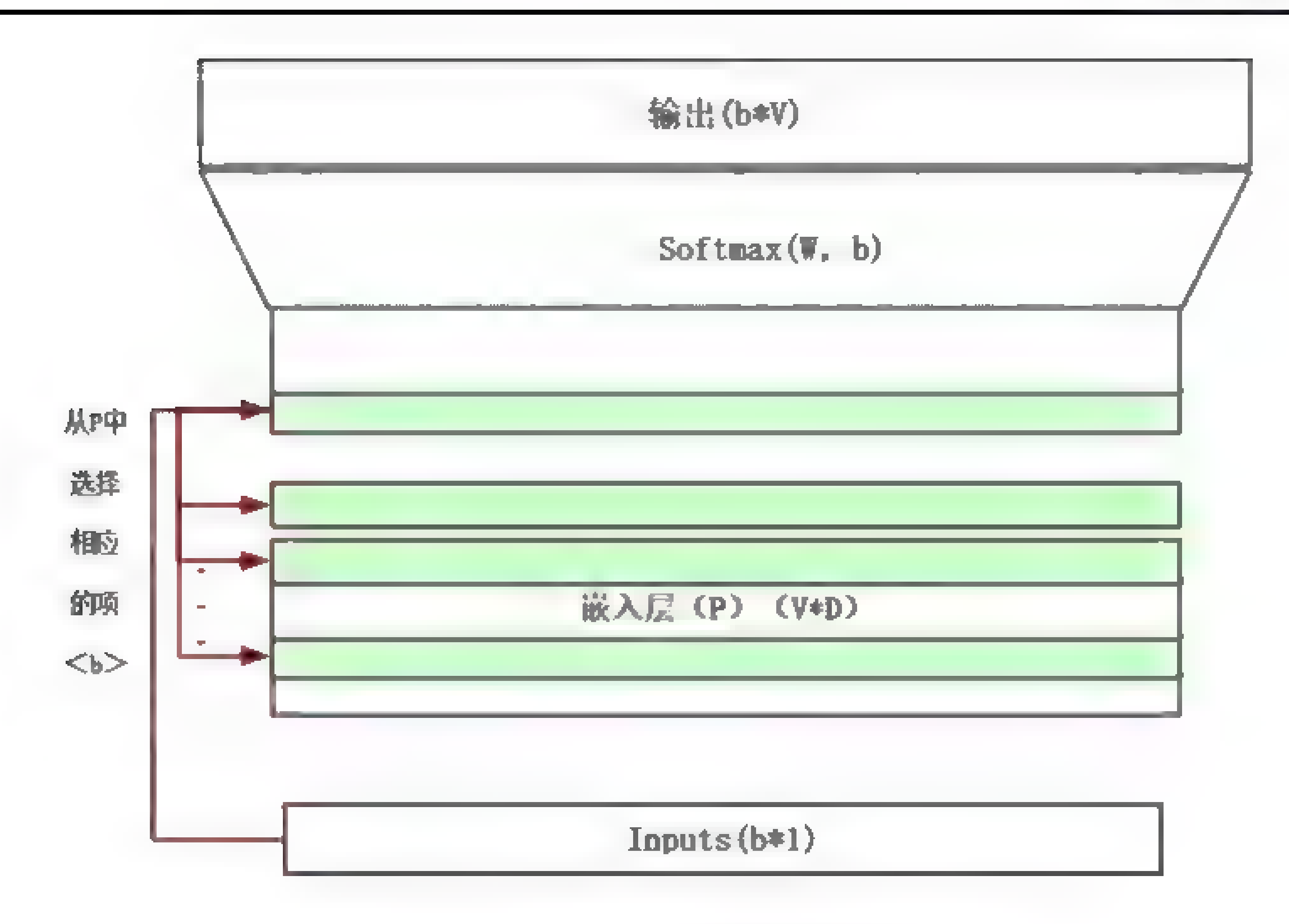


图 4-15 Skip-Gram 模型实施架构图

其中：

- V：词汇量（语料库中的唯一词数）。
- P：投影或词嵌入层（The Projection or the Embedding Layer）。
- D：词向量空间的维数。
- b：单个 batch 的大小。

4.2.6 定义损失函数

到目前为止，我们还没有深入讨论 Word2Vec 的一个非常关键的话题，那就是损失函数。一般情况下，标准 softmax 交叉熵损失函数是分类任务中一个非常好的损失函数。然而，这种损失函数对 Word2Vec 来说有时并不是很实用。在现实工作任务中，可能涉及数十亿个词，词汇量可以轻松达到 100000 个或者更多，这就使得 softmax 函数的归一化变得非常沉重，这是因为 softmax 的完全计算需要计算所有输出节点的交叉熵损失。所以，在保证模型性能的前提下，我们需要转向使用近似且有效的损失函数。这样一来，对于一个 Word2vec 模型设计而言，最大的挑战就变成如何降低 softmax 层的计算复杂度，这也是机器翻译（MT）（Jean 等）和语言建模（Jozefowicz 等）共同关注之处。语言建模中近似 softmax 的方法有多种，例如：

- 多层次 softmax。
- 微分 softmax。
- CNN-softmax。
- 基于采样（Sampling）的方法。
  - ★重要性采样
  - ★具有适应的重要性采样

★目标采样
★噪声对比估计
★负采样
★自标准化
★低频的标准化
★其他方法

下面只讨论一下比较流行的近似选择：负采样和多层次 softmax。

1. softmax 层负采样（Negative Sampling）

对于 softmax 损失函数的替代选择，下面我们将采用更智能的替代方案，称为 sampled softmax 损失。注意，与标准 softmax 交叉熵损失相比，这里有了很多变化。首先，需要计算的是给定目标词所在真实上下文词的 ID 与对应于真实上下文词 ID 的预测值之间的交叉熵损失。其次，我们根据一些噪声分布添加了采样的K 个负样本的交叉熵损失。这就是我们说的对 softmax 层进行负采样。实际数据为（输入 – 输出），噪声为（K-many 虚拟噪声输入 – 输出）。借助于噪声，是指使用不属于目标词所在上下文的词创建的与实际词对（输入 – 输出）不相符的，即使用（K-many 虚拟噪声输入 – 输出）词对。我们还将 softmax 激活函数替换为 sigmoid 激活函数（也称为逻辑函数）。这允许我们在保持[0,1]范围内输出的同时，也可以去除损失函数对整个词汇表的依赖。在较高的层面上，我们将损失定义如下：

$$
\begin{aligned}
Loss &= SigmoidCrossEntropy(Prediction, CorrectWord) \\
&+ \sum_{1}^{K} E_{(NoiseID)} SigmoidCrossEntropy(Prediction, NoiseID)
\end{aligned}
$$

SigmoidCrossEntropy 是我们可以在单个输出节点上定义的损失，与其余节点无关。这使它成为我们分析问题的理想选择，因为我们的词汇量会变得非常大。我们不会深入研究这种损失的细节，也不需要了解这是如何实现的，因为它们可以作为 TensorFlow 中的内置函数使用，但理解损失中涉及的参数（例如 K）很重要。 sampled softmax 损失通过考虑两种类型的实体来计算损失：

- 由预测向量（上下文窗口中的词）中的真实上下文词 ID 给出的索引。
- 词 ID 表示的 K 个索引，被认为是噪声（上下文窗口之外的词）。

负采样是噪声对比估计（Noise-Contrastive Estimation，NCE）方法的近似，根据 NCE 可知，一个好的模型应该通过逻辑回归来区分真实数据和噪声，实际上负采样既很好地保留了模型的性能又很好地做到有效损失的近似。

结合上面的例子来说明这一点，如图 4-16 所示。

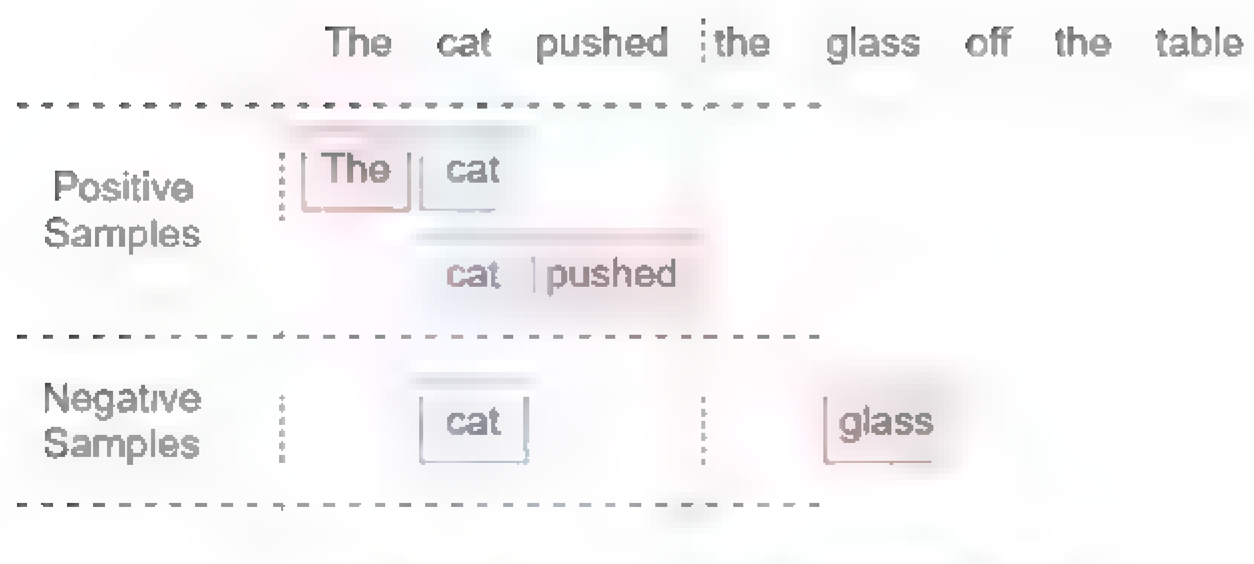


图 4-16 softmax 层负采样示例

2. 多层次 softmax（Hierarchical softmax）

多层次 softmax（H-softmax）是 Morin 和 Bengio 受到二叉树启发而得出的方法。从根本上来说，H-softmax 是用词语作为叶子的多层结构来替代原 softmax 的一层，如图 4-17 所示。

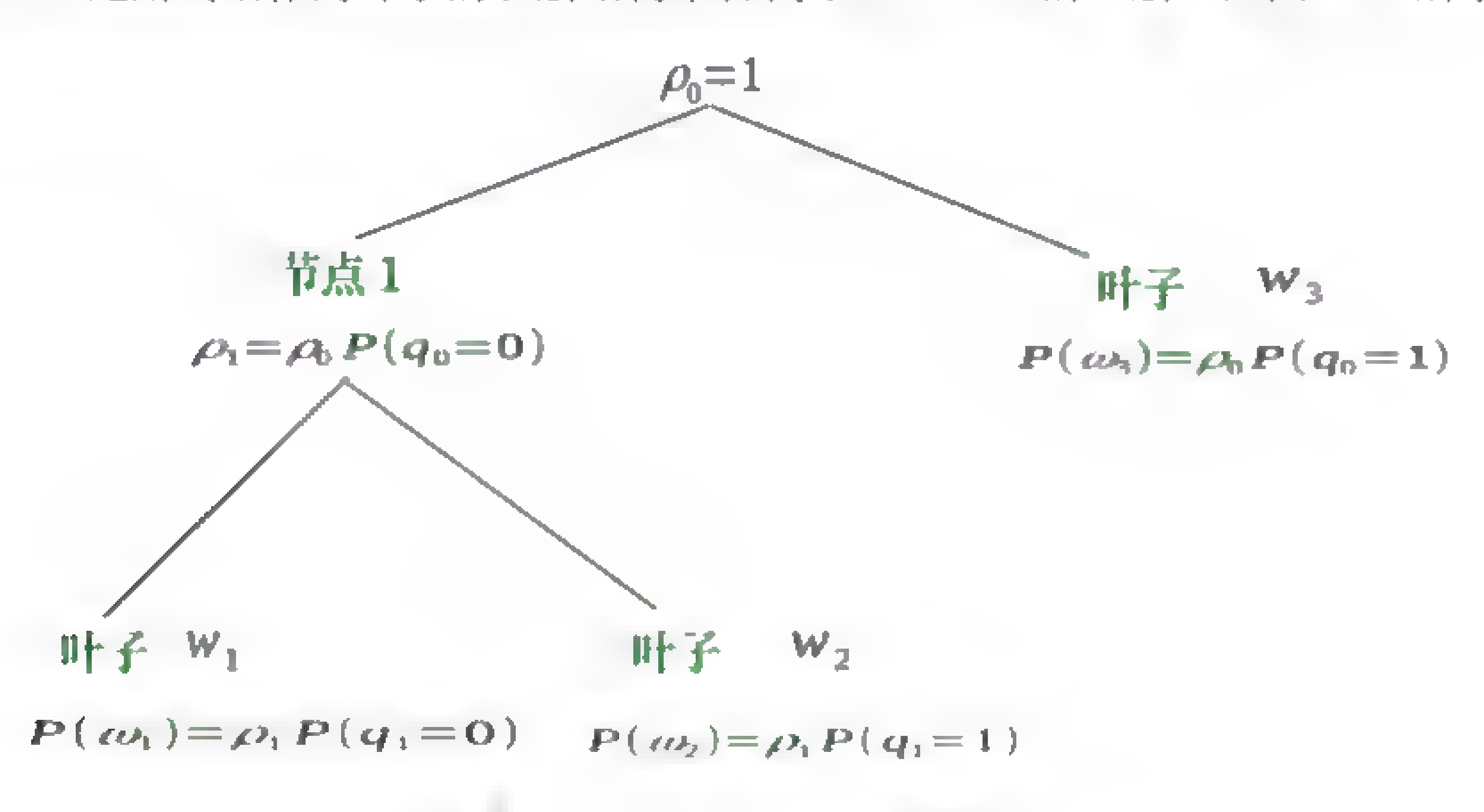


图 4-17 多层次 softmax 图示

这样一来，对于单个词出现概率的计算就可以被分解为一连串的概率计算，我们也就无须再对所有词进行高成本的标准化计算处理。用 H-softmax 来替代单一的 softmax 层可以大大提升预测词的速度，有研究表明，至少带来 50 倍的提升，因此特别适用于要求低延迟的工作任务，比如谷歌的新通信软件 Allo 的实时沟通功能便是如此。

我们如果把常规的 softmax 层想象成只有一层的树，每个 V 中的词均是一个叶子节点，那么在计算一个词 softmax 层的概率时，就需要标准化所有|V|个叶子的概率，显然计算成本非常高。如果把 softmax 层当成每个词都是叶子的一棵二叉树，则只需要从叶子节点开始沿着树的路径行走，就可以抵达指定的词，而无须考虑其他词，显然这种方法更佳。

多层次 softmax 比负采样略微复杂，但与负采样的目标相同。与负采样的不同之处是多层次 softmax 仅使用实际数据而不需要噪声。我们通过一个例子来做进一步解读。例如，有下面一句话：

```
I like NLP. Deep learning is amazing.
```

上面这个句子的词汇如下：

```
I, like, NLP, Deep, learning, is, amazing
```

使用这个词汇表，可以构建一棵二叉树，其中词汇表中的所有词都以叶子节点的形式出现。我们还将添加一个特殊的标记 PAD，以确保所有树叶都有两个成员，如图 4-18 所示。

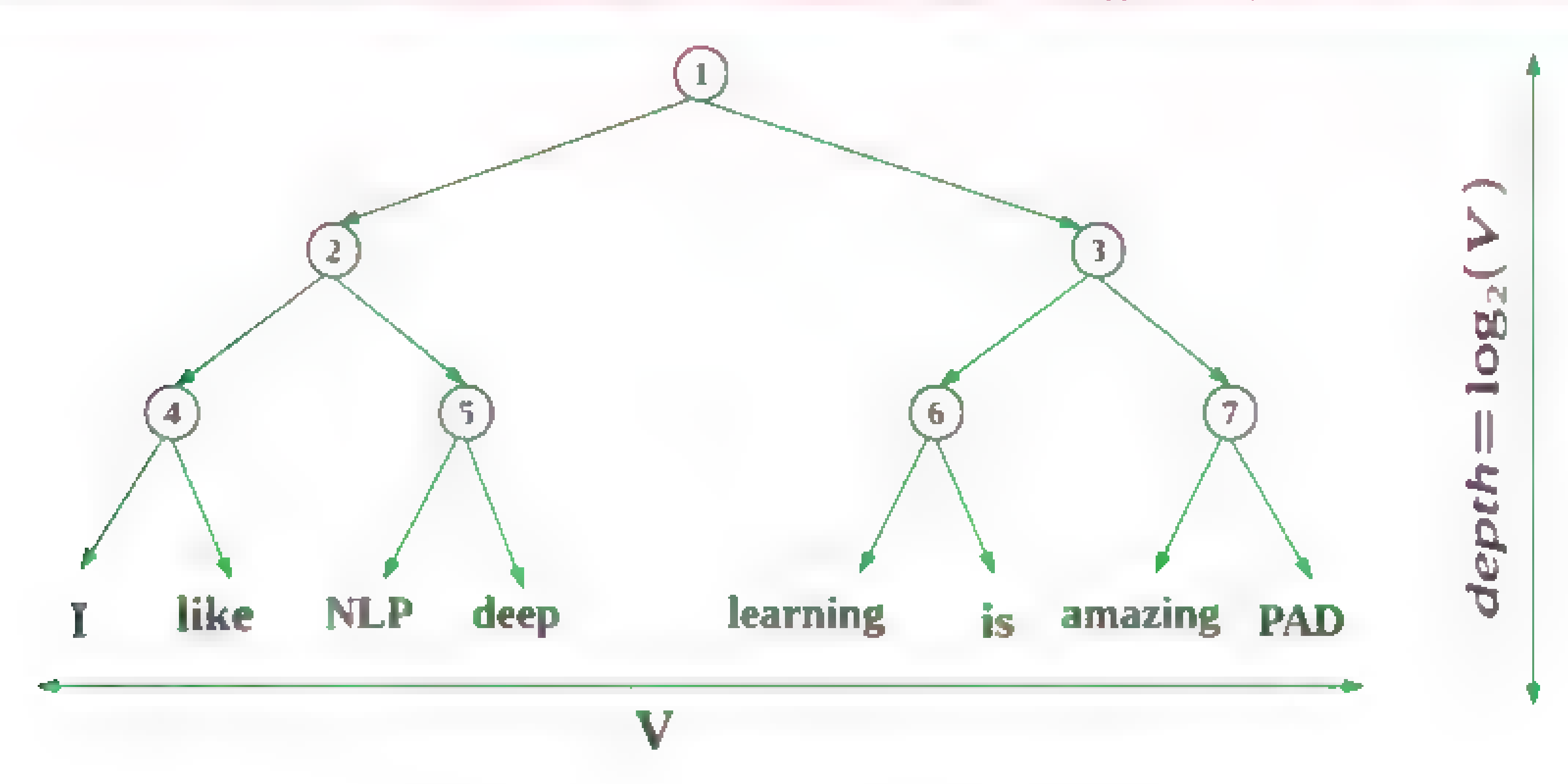


图 4-18　所有词以叶子节点出现

接着，最后一个隐藏层将完全连接到层次结构中的所有节点。这里与经典的 softmax 层相比，该模型具有相似的总权重值，但是，对于给定的计算，它仅使用其中一部分。

如果我们要计算 P（NLP | like）的概率，只需要一个权重值子集来计算概率即可，其中 like 是输入词，如图 4-19 所示。

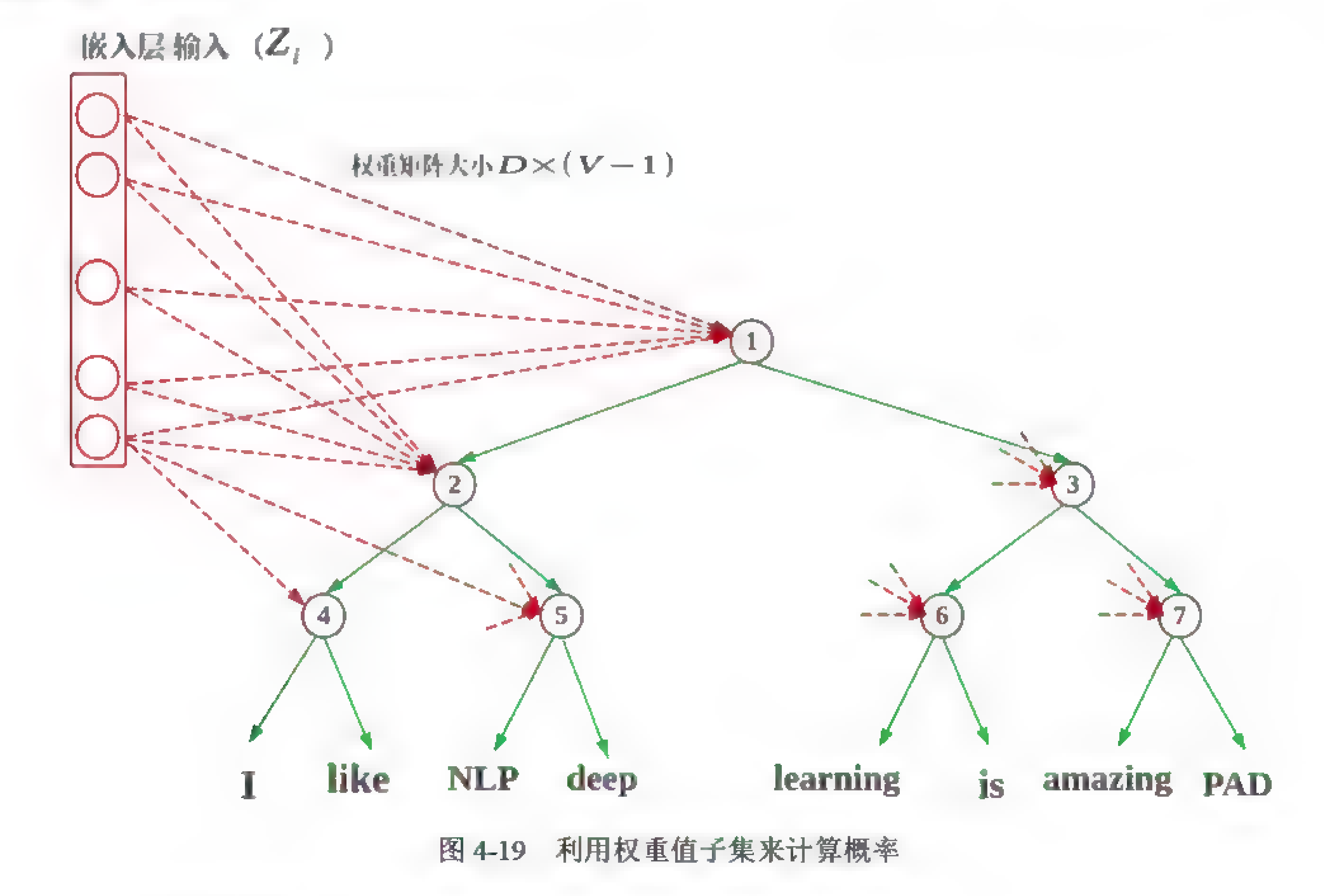


图 4-19　利用权重值子集来计算概率

具体来说，计算概率如下：

```
( NLP | like ) = P ( left at 1| like ) × P ( right at 2 | like ) × P ( left
at 5 | like )
```

由于知道了如何计算 P()，我们便可以使用原始损失函数。注意，该方法仅使用连接到路径中节点的权重值进行计算，从而使得计算效率很高。

尽管多层次 softmax 是有效的，但是一个重要的问题仍然值得注意，那就是如何确定树的分解。更准确地说，哪个词会跟随哪个分支。下面给出两个解决方案。

（1）随机初始化层次结构：事实上，此方法会使模型的一些性能下降，因为无法保证随机分配在词之间的分支是最佳的。

（2）使用 WordNet 确定层次结构：WordNet 可用于确定树中词的合适顺序。该方法明显表现出比随机初始化更好的性能。

4.2.7　利用 TensorFlow 实现 Skip-Gram 模型

接下来我们将使用 TensorFlow 来实现 Skip-Gram 算法。在这里，我们将仅仅讨论涉及定义 TensorFlow 的操作以便进行学习词向量的部分。完整代码可在 ch4 文件夹下的 4_skip-gram_CBOW(improved).ipynb 中找到。

这里下载的数据集包含多个维基百科文章，总计大约 61MB。数据集来源见链接 http://www.evanjones.ca/software/wikipedia2text.html。

首先定义模型的超参数。你可以自由更改这些超参数的值以查看它们如何影响最终的性能（例如，batch_size = 16 或 batch_size = 256）。具体如下：

```
batch_size = 128
embedding_size = 128 # 词向量的维数
window_size = 4 #左右两边各考虑多少个词
valid_size = 16 #用于评估相似性的随机词集
# 仅在分布的头部选择开发样本
valid_examples = np.array(random.sample(range(valid_window), valid_size))
valid_examples = np.append(valid_examples,random.sample(range(1000,
1000+valid_window), valid_size),axis=0)
num_sampled = 32 #要抽样的负样例数量
```

接下来，为训练输入数据集、标签和有效输入定义 TensorFlow 占位符：

```
train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

然后，为词嵌入层和 softmax 层的权重值及偏差定义 TensorFlow 变量：

```
embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size],
-1.0, 1.0))
```

```
    softmax_weights = tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size], stddev=0.5 / math.sqrt(embedding_size)))
    softmax_biases = tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01))
```

接下来，我们将定义一个词向量查找操作，该操作收集指定训练数据集对应的词向量：

```
    embed = tf.nn.embedding_lookup(embeddings, train_dataset)
```

之后，我们将使用负采样来定义 softmax 损失：

```
    loss = tf.reduce_mean(tf.nn.sampled_softmax_loss(weights=softmax_weights,
    biases=softmax_biases,inputs=embed, labels=train_labels,
num_sampled=num_sampled, num_classes=vocabulary_size))
```

在这里，我们定义一个优化器来优化（最小化）前面定义的损失函数。我们也可以尝试使用 https:// tensorflow.google.cn /api_guides/python/train 中列出的其他优化器进行试验：

```
    optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

计算验证输入词示例和所有词向量之间的相似性。使用余弦距离：

```
    norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keepdims=True))
    normalized_embeddings = embeddings / norm
    valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,
valid_dataset)
    similarity = tf.matmul(valid_embeddings,tf.transpose(normalized_embeddings))
```

在定义了所有 TensorFlow 变量和操作后，现在可以继续执行一些操作，这里会简要给出这些操作的基本过程，具体的完整过程请参见对应的代码文件。

（1）使用 tf.global_variables_initializer()初始化 TensorFlow 变量.run()。

（2）对于每个步骤（预定义的总步骤数），请执行以下操作：

使用数据生成器生成一批数据（batch_data - inputs，batch_labels -outputs）。

创建一个名为 feed_dict 的字典，将训练输入/输出占位符映射到数据生成器生成的数据：

```
    feed_dict = {train_dataset: atch_data, train_labels: batch_labels}
```

执行优化步骤并获取损失值，如下所示：

```
    _, l = session.run ([optimizer, loss], feed_dict = feed_dict)
```

最终，Skip-Gram 模型借助 t-SNE 技术对于相关数据进行可视化后的效果如图 4-20 所示。

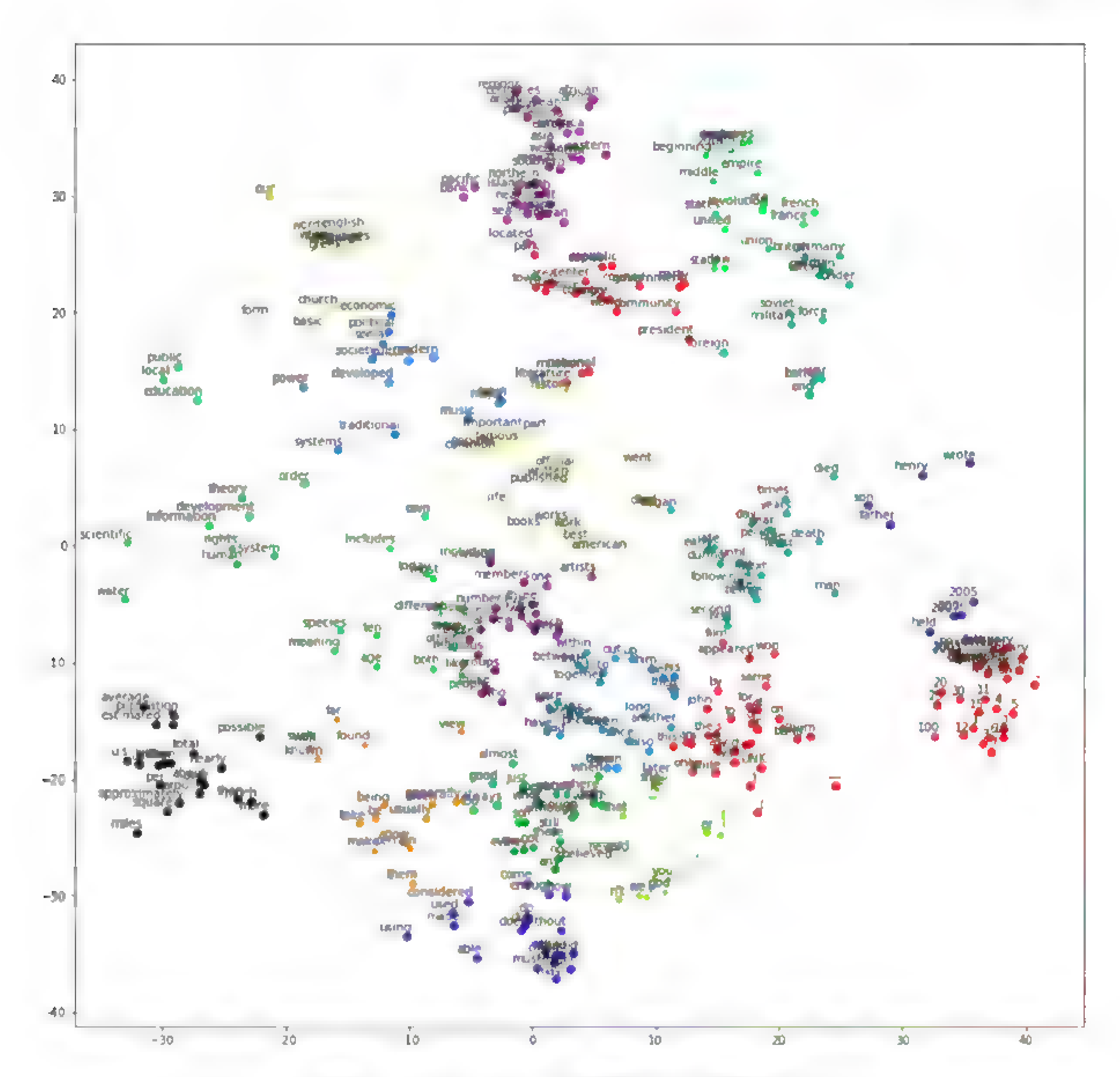


图 4-20　Skip-Gram 模型可视化效果图示

将图 4-20 中的部分结果放大，如图 4-21 所示。

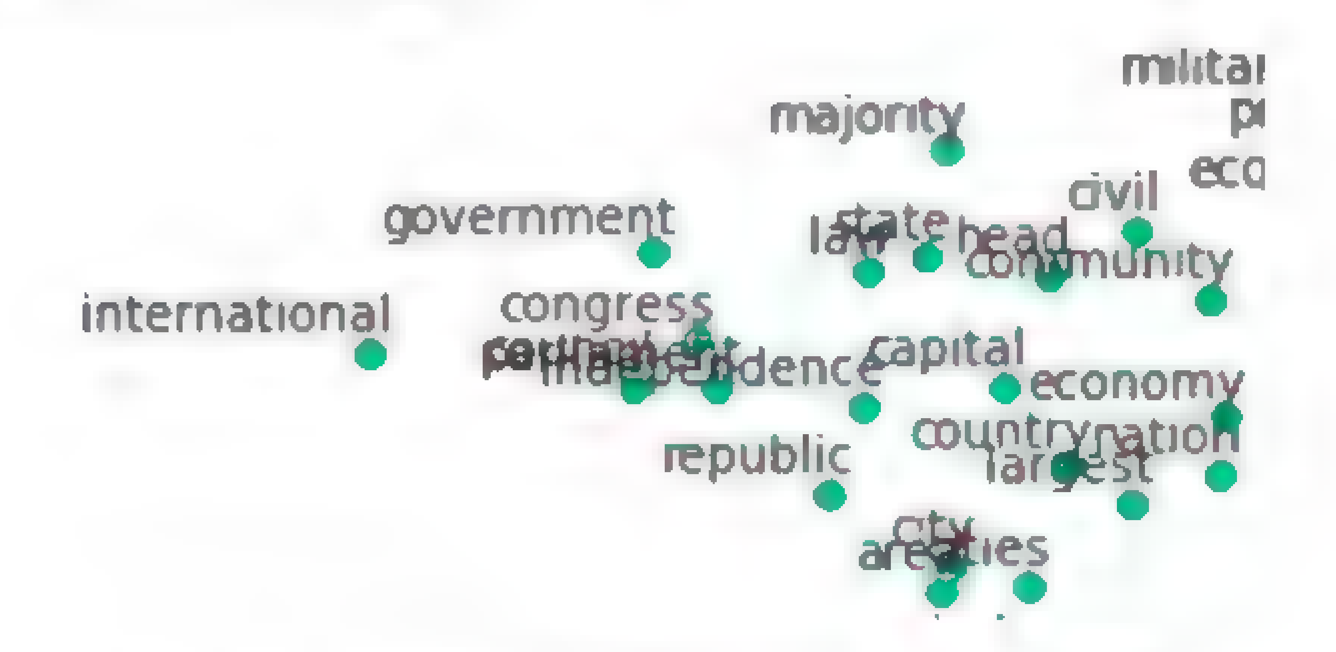


图 4-21　Skip-Gram 模型可视化部分放大效果图示

从放大的结果来看，这样的分类结果还是符合预期的，其余部分的分类结果，读者可以自行查验。

4.3 原始 Skip-Gram 模型和改进 Skip-Gram 模型对比分析

到目前为止，讨论的 Skip-Gram 算法实际上是对 Mikolov 等人在 2013 年出版的原始论文中提出的原始 Skip-Gram 算法的改进模型。在本节中，我们先看一下原始的 Skip-Gram 算法。

原始 Skip-Gram 算法在进行表示学习时，没有使用中间隐藏层，所以在代码实现过程中不会设置 softmax_weights 和 softmax_biases 等神经网络参数，因为它只有输入层和输出层，如图 4-22 所示，并定义了一个从词向量本身派生的损失函数。

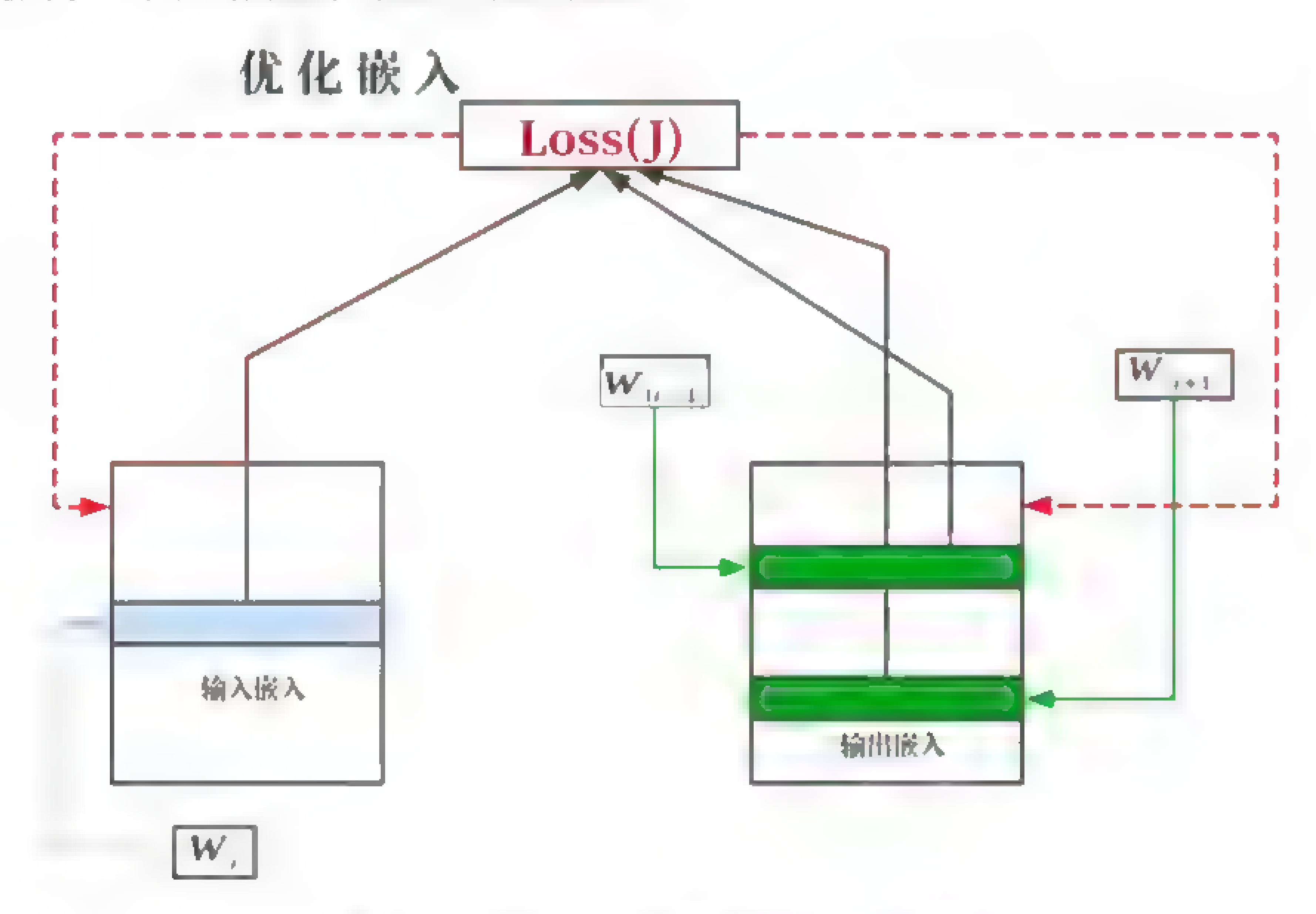


图 4-22 不含隐藏层的原始 Skip-Gram 算法

对于原始 Skip-Gram 算法而言，其负采样损失定义如下：

$$J(\theta) = -\left(\frac{1}{(N-2m)}\right)\sum_{(i=m+1)}^{(N-m)}\sum_{(j\neq i\wedge j=i-m)}^{(i+m)} log\left(\sigma\left(v'_{(w_j)}v_{(w_i)}\right)\right) + kE_{((w_q)\sim(P_n(w)))}\left[log\sigma\left(-v'_{(w_q)}v_{(w_i)}\right)\right]$$

这里，v 是输入词嵌入层，v' 是输出词嵌入层，$v_{(wi)}$ 对应于输入词嵌入层中词 w_i 的词向量，而 $v'_{(w_i)}$ 对应于输出词嵌入层中词 w_i 的词向量。$P_n(w)$ 是噪声分布，我们从中对噪声样本进行采样。最后，E 表示从 k-负样例获得的损失期望（平均值）。正如我们看到的，除了词向量本身之外，此等式中没有权重值和偏差。

4.3.1　原始的 Skip-Gram 算法的实现

原始的 Skip-Gram 算法在计算损失函数时，需要使用 TensorFlow 函数手工完成，因为其中缺少内置函数来计算损失。完整代码见 ch4 文件夹下 4 comparison.ipynb 文件。

首先，为输入数据集和输出数据集各定义一个占位符：

```
train dataset = tf.placeholder(tf.int32, shape=[batch size])
train labels = tf.placeholder(tf.int64, shape=[batch size, 1])
```

通过定义输入和输出占位符，我们可以调用 TensorFlow 内置函数 candidate_sampler 进行负采样，代码如下所示：

```
negative_samples, _, _ = tf.nn.log_uniform_candidate_sampler(
                                                  train_labels, num_true=1,
                                                  num_sampled=num_sampled,
                                                  unique=True,
                                                  range_max=vocabulary_size)
```

在这里，我们统一对负样例进行抽样。接着，我们有 num_true 的数量，它表示给定数据点的真实类的数量。接下来是我们想要一批数据的负样本数（num_sampled），而 unique 参数来定义负样本是否应该是唯一的。最后，range_max 定义一个词具有的最大 ID，使得采样时避免采到无效词 ID。

模型除去 softmax 的权重值和偏差之后，我们引入两个词嵌入层，一个用于输入数据，另一个用于输出数据。（这里需要两个词嵌入层，因为只有一个词嵌入层，损失函数将不起作用。）

接下来，将编写代码中最重要的部分——定义损失函数。这段代码实现了我们前面讨论过的损失函数。但是，我们不会立即计算文档中所有词的损失，因为如果文档太大的话可能会引起内存溢出。因此，我们将会计算单个时间步长内的小批量数据的损失。完整代码可在 ch4 文件夹中的 4_word2vec_improvements.ipynb 文件中找到：

```
# 计算正样本的损失
loss = tf.reduce_mean(
      tf.log(
          tf.nn.sigmoid(
                     tf.reduce_sum(
                            tf.diag([1.0 for _ in range(batch_size)])*
                            tf.matmul(out_embed,tf.transpose(in_embed)),
                      axis=0)
          )
     )
)
#计算负样本的损失
loss += tf.reduce mean(
      tf.reduce sum(
```

```
    tf.log(tf.nn.sigmoid( tf.matmul(negative_embed,tf.transpose(in_embed)))),
              axis=0
         )
)
```

注意

TensorFlow 通过定义较小的权重值和偏差子集来实施 sampled softmax loss，这些权重值和偏差仅需要处理当前批量数据即可。此后，TensorFlow 计算类似于标准 softmax 交叉熵计算的损失，但是由于没有 softmax 权重值和偏差，因此我们无法直接转换为该方法来计算原始的 Skip-Gram 损失。

4.3.2 将原始 Skip-Gram 与改进后的 Skip-Gram 进行比较

通过代码的运行，我们针对原始 Skip-Gram 与改进后的 Skip-Gram 算法在计算损失效果上进行了对比，详见图 4-23。

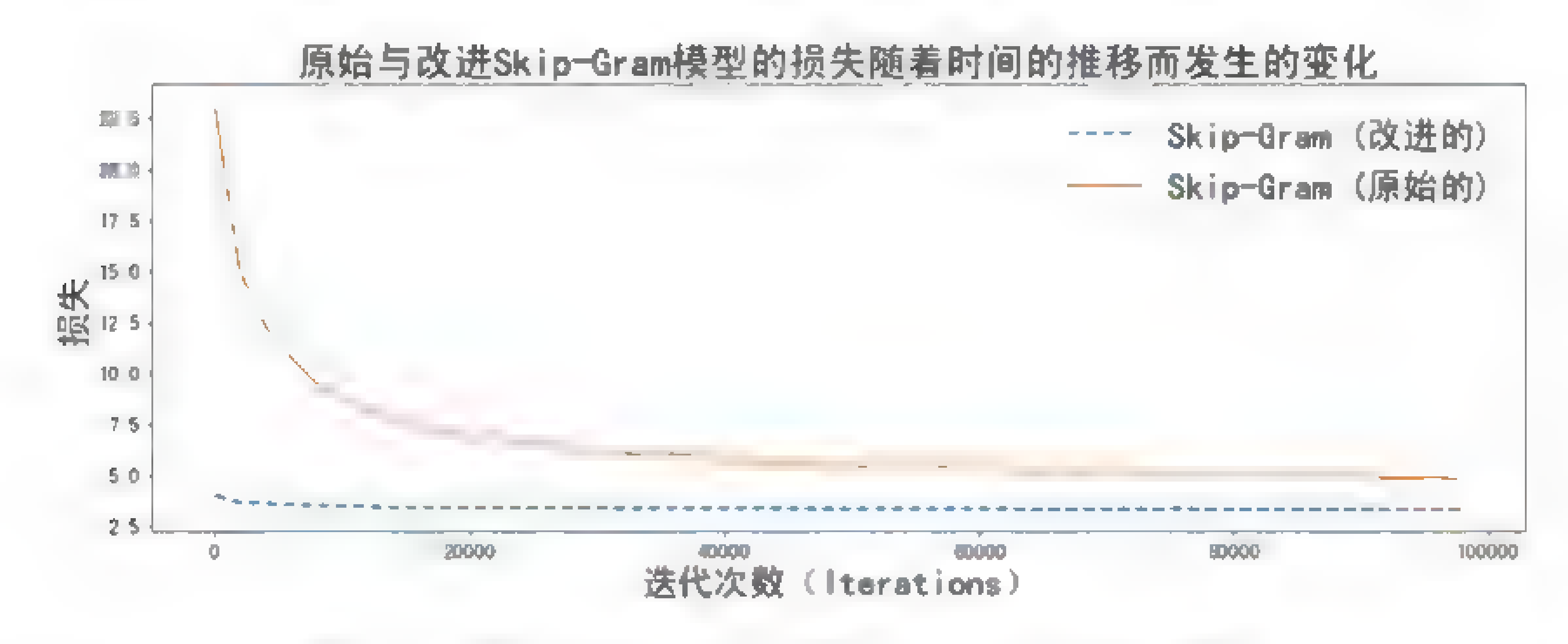


图 4-23 原始 Skip-Gram 与改进 Skip-Gram 的损失变化

我们可以清楚地看到，与没有隐藏层相比，拥有隐藏层的 Skip-Gram 算法会带来更好的性能。这也表明拥有更深层神经网络结构的 Skip-Gram 模型往往表现更好，当然对于 Word2vec 模型而言也是同样的道理。

4.4 CBOW 模型

4.4.1 CBOW 模型简述

这里再简要提一下 Word2vec 模型结构。Word2vec 是使用单个隐藏层且完全连接的神经网络，如图 4-24 所示。其中，隐藏层中的神经元都是线性神经元，输入层设置了与用于训练的词汇中词一样多的神经元，隐藏层大小与生成词向量的维度一致，且输出层的大小与输入层相同。因此，假设用于学习词向量的词汇表由 V 个词组成并且词向量的维度为 N，则隐藏层连接的输入大小可以

用 $V\times N$ 的矩阵 WI 表示，每行表示一个词；同理，可以用 $N\times V$ 的矩阵 WO 来描述从隐藏层到输出层的连接。在这种情况下，WO 矩阵的每列表示来自给定词汇表的词。这里使用 One-hot 编码（独热编码）对于所有的输入进行编码。

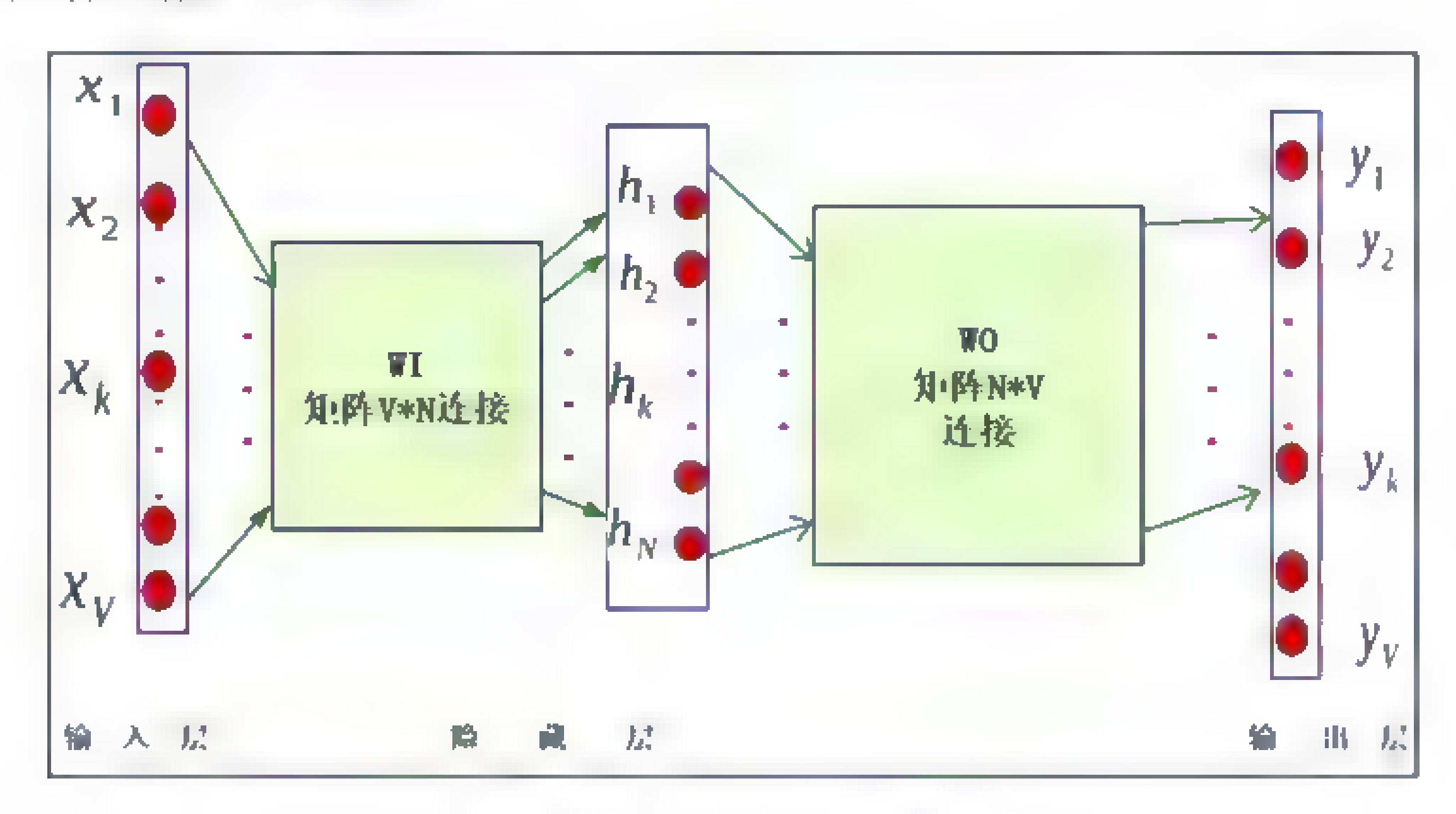


图 4-24　Word2vec 概念图

在 Skip-Gram 模型中，我们预测了目标词中的上下文词，但是，在 CBOW 模型中，我们将从上下文词预测目标词。让我们通过下面的句子来比较 Skip-Gram 和 CBOW：

```
The dog barked at the mailman.
```

对于 Skip-Gram，数据元组(输入词,输出词)，可以表示为(dog, the)、(dog, barked)、(barked, dog)等。

对于 CBOW，数据元组可以表示为：([the, barked], dog)、([dog, at], barked)等。

因此，CBOW 的输入具有 $2\times m\times D$ 的维度，其中 m 是窗口大小、D 是词向量的维度。CBOW 的概念模型如图 4-25 所示。

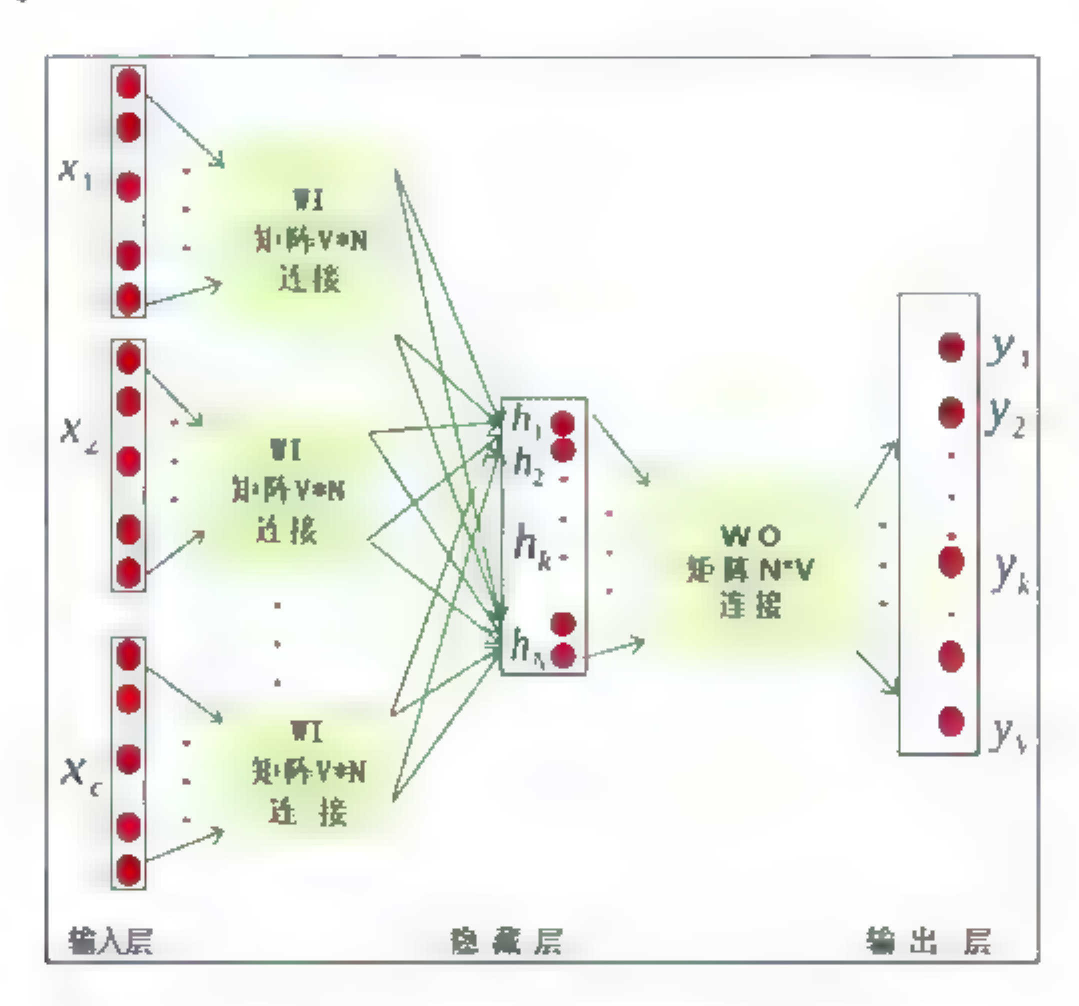


图 4-25　CBOW 概念结构示意图

在图 4-25 中，隐藏层的输出是输入层处对应上下文词向量的平均向量。由于 CBOW 模型与

Skip-Gram 模型非常相似，因此这里就不再做过多解读。下面，我们将通过 TensorFlow 来实现 CBOW 算法。CBOW 的完整实现可以在 ch4 文件夹的 4 skip-gram CBOW(improved).ipynb 中找到。

4.4.2 利用 TensorFlow 实现 CBOW 算法

首先，我们定义相关变量，这与 Skip-Gram 模型的情况相同：

```
    #词嵌入层
    embeddings = tf.Variable(tf.random uniform([vocabulary size,embedding_size],
-1.0, 1.0, dtype=tf.float32))
    #softmax 的权重值和偏差
    softmax_weights = tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size],stddev=1.0 /

math.sqrt(embedding_size),dtype=tf.float32))

    softmax_biases =tf.Variable(tf.zeros([vocabulary_size],dtype=tf.float32))
```

在这里，我们创建了一组叠加的词向量，代表了上下文的每个位置。所以我们将有一个矩阵：[batch_size，embeddings_size，2 * context_window_size]。然后，我们将使用简化运算符对最后一个轴上的堆叠词向量进行平均，以便将堆叠矩阵减小为 [batch_size, embedding size]：

```
    stacked_embedings = None
    for i in range(2*window_size):
        embedding_i = tf.nn.embedding_lookup(embeddings,train_dataset[:,i])
        x_size,y_size = embedding_i.get_shape().as_list()
        if stacked_embedings is None:
            stacked_embedings = tf.reshape(embedding_i,[x_size,y_size,1])
        else:
            stacked_embedings
=tf.concat(axis=2,values=[stacked_embedings,tf.reshape(embedding_i,
                                                    [x_size,y_size,1])])
    assert stacked_embedings.get_shape().as_list()[2]==2*window_size
    mean_embeddings = tf.reduce_mean(stacked_embedings,2,keepdims=False)
```

此后，像 Skip-Gram 模型一样去定义损失函数和优化器：

```
    loss = tf.reduce_mean(
    tf.nn.sampled_softmax_loss(weights=softmax_weights,biases=softmax_biases,in
puts=mean_embeddings,
    labels=train_labels,num_sampled=num_sampled,num_classes=vocabulary_size))
    optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

运行 CBOW 模型后，最终得到的结果如下（部分展示）：

```
第 94000 步长上的平均损失: 2.087824
第 96000 步长上的平均损失: 2.111227
第 98000 步长上的平均损失: 2.094030
第 100000 步长上的平均损失: 2.101904
与 that 最接近的单词: which, how, what, heroes, jogaila, pro, acoustics, taxa,
与 but 最接近的单词: however, although, though, which, patriots, calories,
internationalist, thracian,
与 or 最接近的单词: and, 3,900, solidifying, and/or, fictions, branching, than,
afl-cio,
与 this 最接近的单词: it, monna, 1786, mesons, another, 2.25, aptera,
archaefructus,
与 the 最接近的单词: macflecknoe, joaquim, spellings, separator, a, tcb, heikki,
elision,
与 's 最接近的单词: ' , s, calabash, arguably, transformative, his, pikes, louis,
与 in 最接近的单词: throughout, until, marie, since, axles, outside, during,
three-tiered,
```

这样看来，效果还不错。具体结果，读者可以自行运行代码去查看。

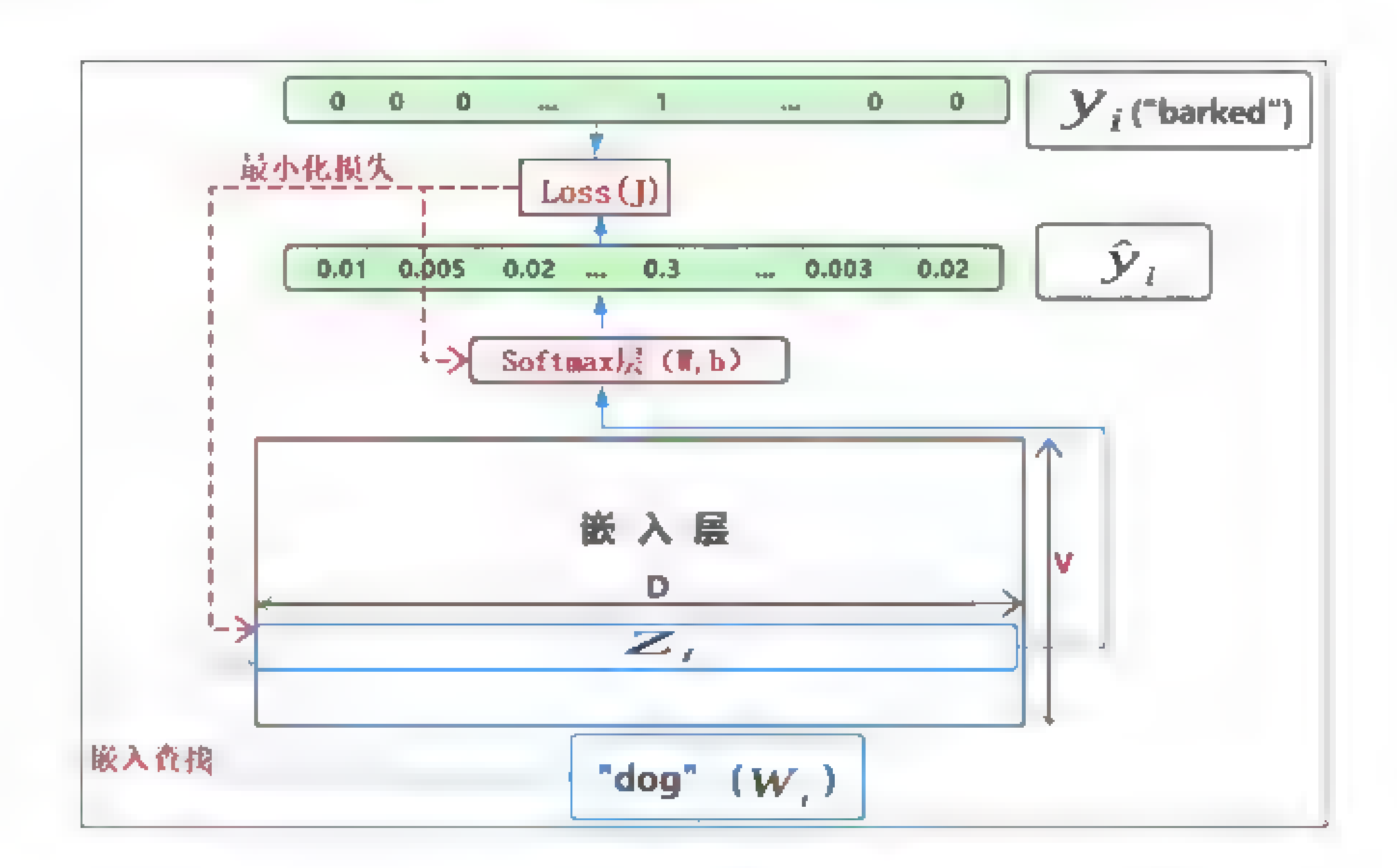
4.5 Skip-Gram 和 CBOW 对比

在介绍完 Word2vec 的两个模型之后，下面我们分析一下这两个模型之间的差异。

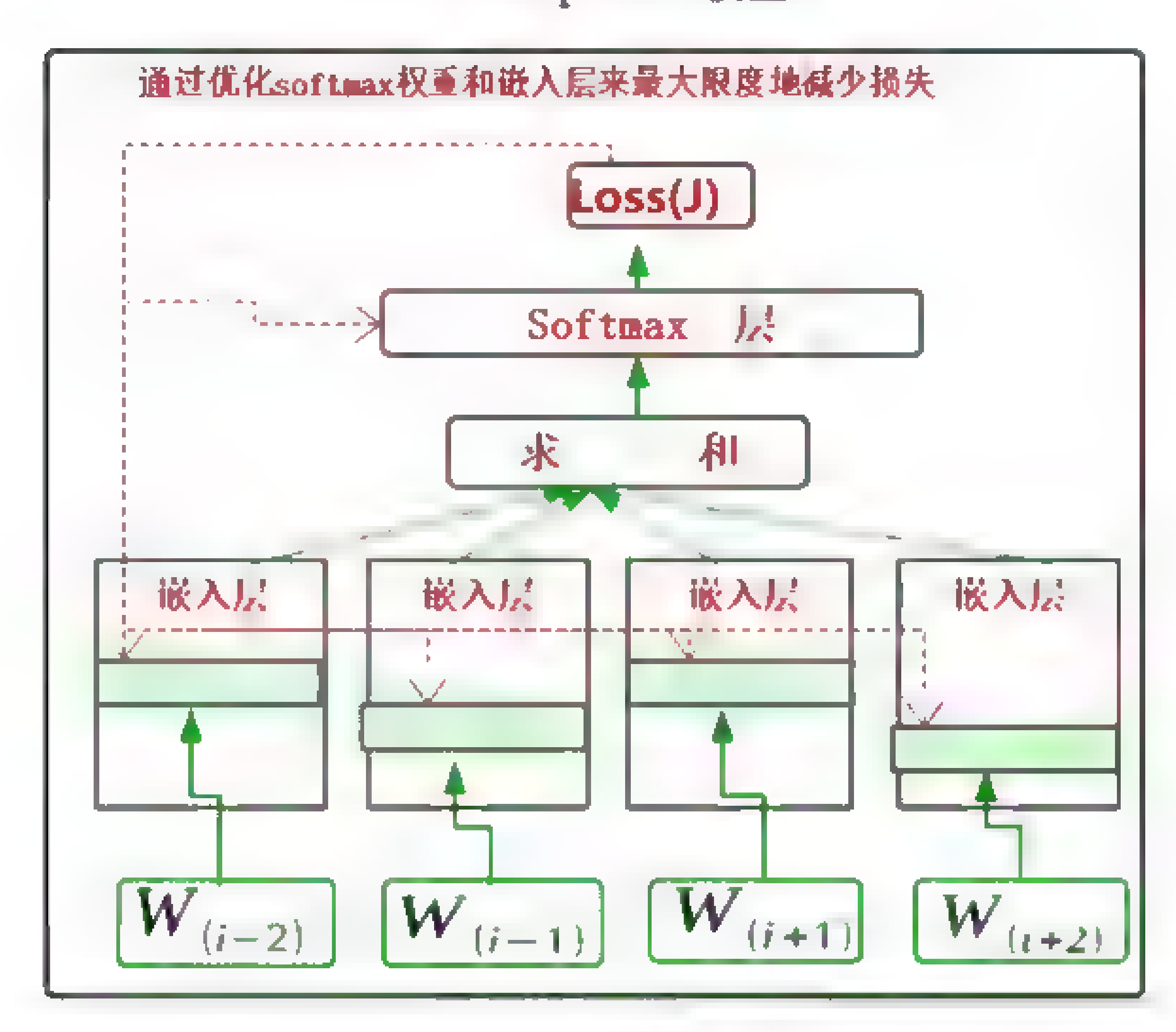
4.5.1 Skip-Gram 和 CBOW 模型结构分析

这里结合 4.4.1 中的例句（如图 4-26 所示）给定上下文和目标词，Skip-Gram 模型其实仅在单个[输入,输出]元组中关注目标词和上下文中的单个词，而 CBOW 在单个样本中则关注目标词和上下文中的所有词。例如，短语 dog barked at the mailman 中，Skip-Gram 在单个时间步长内关注到像["dog","at"]这样的输入输出元组，而 CBOW 关注到的则是[["dog","barked","the","mailman"],"at"]这样的输入输出元组。所以，在给定的一批数据中，CBOW 接收的信息多于 Skip-Gram。接下来我们了解一下这种差异如何影响两种算法的性能。

所以，从两个模型的实现视图来看，我们知道，CBOW 模型在给定的时间里可以获取更多的信息（输入），这样就促使 CBOW 在某些条件下性能更优。



（a）Skip-Gram 模型



（b）CBOW 模型

图 4-26 实现视图

4.5.2 代码层面对比两模型性能

在训练模型的任务时，我们通过代码绘制了 Skip-Gram 和 CBOW 中随着迭代次数变化而产生的损失情况，如图 4-27 所示，完整代码详见 ch4 文件夹中的 4 comparison.ipynb 文件。

显然，我们可以发现，随着迭代次数的堆叠，CBOW 模型的损失比 Skip-Gram 模型的损失下降的速度快很多。但是，就模型的性能衡量而言，只关注损失本身是不够的，因为如果模型过度拟合了训练数据，损失照样会迅速减少，所以我们还要关注模型的另外一个性能指标，即词向量的质量。为了更加直观地比较 Skip-Gram 和 CBOW 模型之间的性能情况，这里使用 t-SNE 技术进行可视化对比，其运行结果如下，读者可以自行执行代码进行查验。

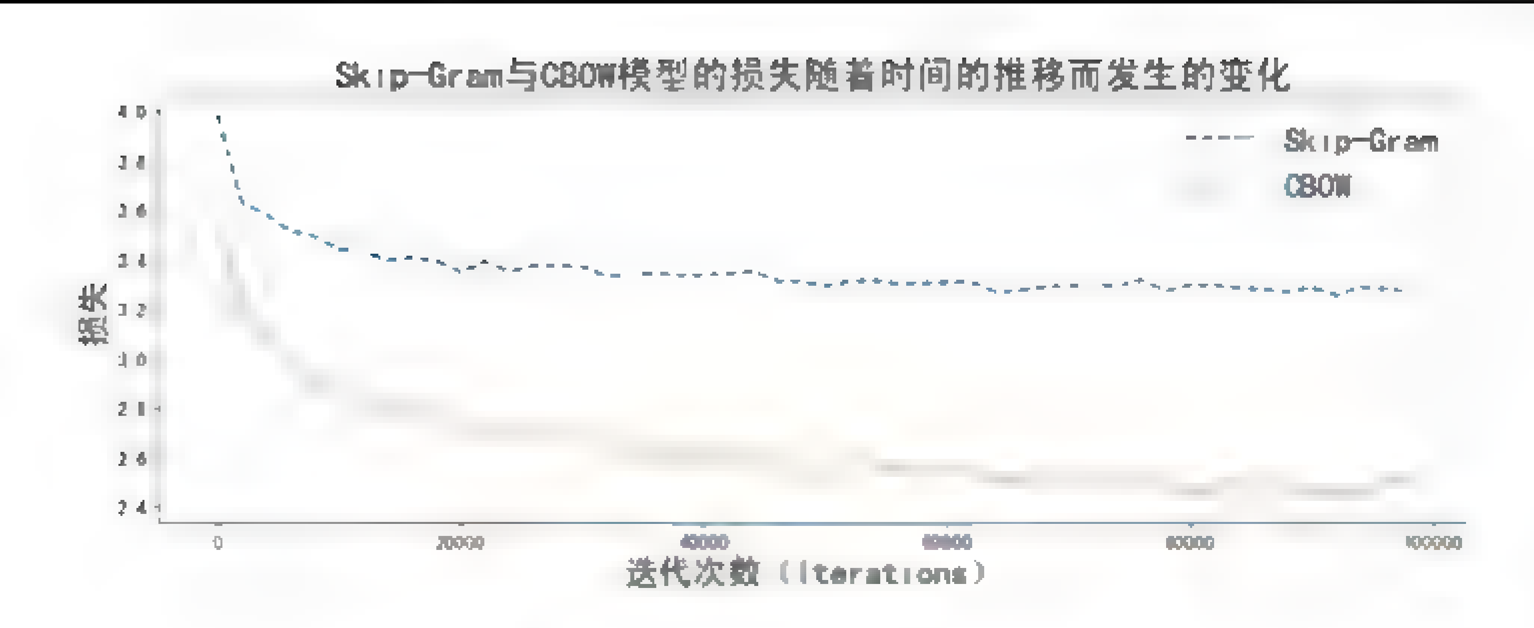


图 4-27　损失下降: Skip-Gram vs CBOW

我们讨论过，与 Skip-Gram 算法相比，CBOW 可以访问给定[输入,输出]元组的指定目标词所在上下文的更多信息。可以看到，与 Skip-Gram 模型相比，CBOW 给出的损失会快速减少。尽管如此，其实损失本身并不足以衡量模型绩效，因为过度拟合了训练数据，损失也会迅速减少。虽然有一些基准测试任务用于评估词向量的质量（例如，词类比任务），但我们将使用更简单的检查方法。为了直观地检查学习的词向量，以确保 Skip-Gram 和 CBOW 在它们之间显示出显著的语义差异，这里使用 t-Distributed 随机邻居嵌入（t-SNE）可视化技术进行处理，详见图 4-28（详见代码运行效果图）。在图 4-28 中，我们可以看到 CBOW 对于词的聚集效果优于 Skip-Gram，其中词稀疏地分布在整个空间中。因此，我们可以说 CBOW 看起来比 Skip-Gram 更具有优势，然而，事实上这仅是一个特例，下面会继续分析。

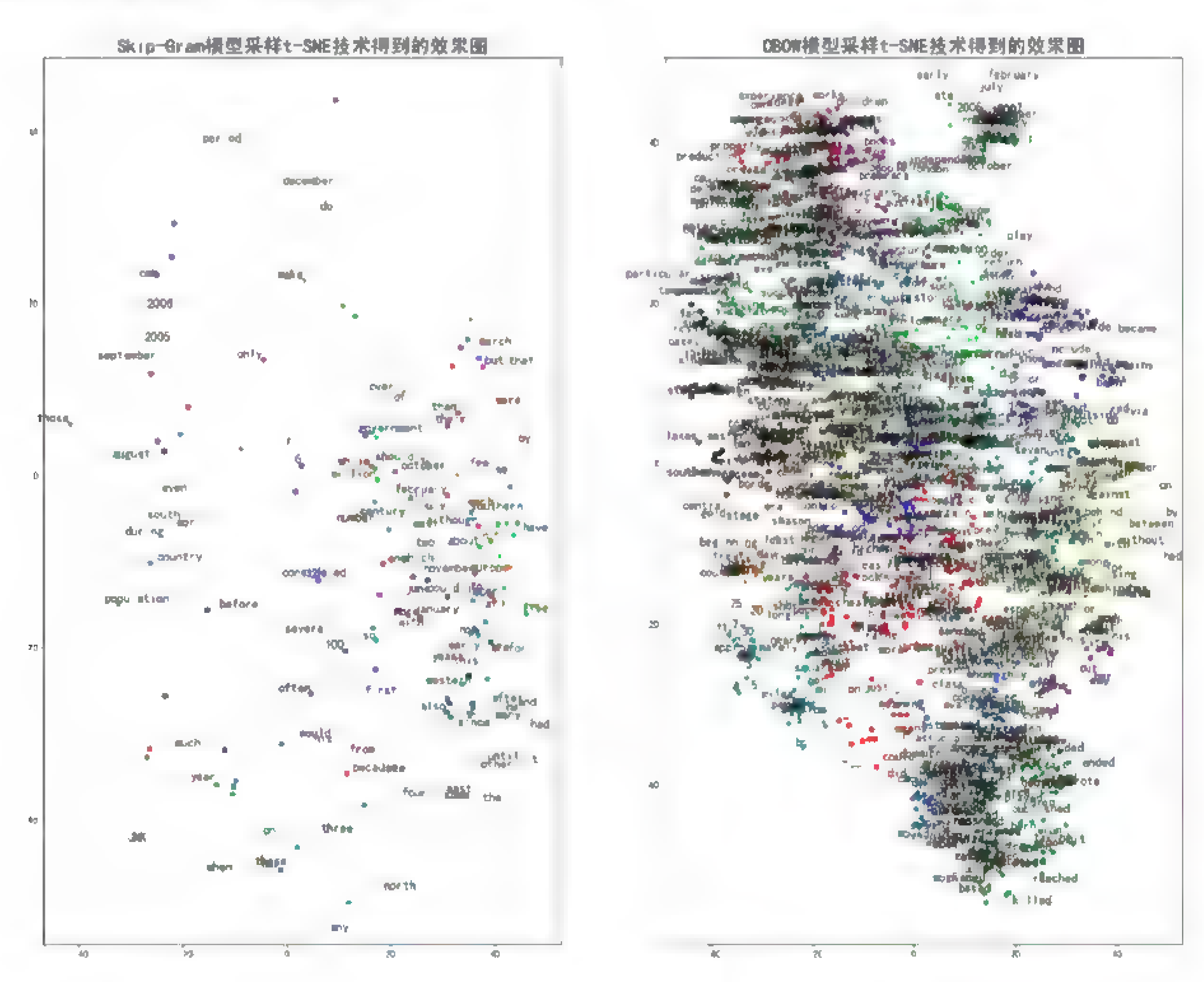


图 4-28　Skip-Gram 和 CBOW 获得的词向量的 t-SNE 可视化效果图示

4.5.3 Skip-Gram 和 CBOW 模型孰优

其实，在性能表现方面，Skip-Gram 和 CBOW 之间没有明确的赢家。就像 Mikolov 等人在 2013 年发表的论文《*Distributed Representations of Words and Phrases and their Compositionality*》（Mikolov 等，2013）中所表达的那样，Skip-Gram 在语义任务中表现更好，而 CBOW 在句法任务中表现更好。但是，在大多数任务中，Skip-Gram 似乎比 CBOW 表现更好，这与我们刚才的发现显然有些出入。

各种经验证据表明，与 CBOW 相比，Skip-Gram 适用于大型数据集，并且在论文《*Distributed Representations of Words and Phrases and their Compositionality*》（Mikolov 等，2013）和论文《*GloVe: Global Vectors for Word Representation*》（Pennington 等，2014）中得到了支持，这里通常使用数十亿个词的语料库。由于上面任务涉及的数据只有数十万个词，数据样本相对较小，所以这时 CBOW 表现更好。

为了更好地证实上面的论述，下面我们做进一步分析，考虑以下两句话：

- It is a nice day
- It is a brilliant day

对于 CBOW，输入输出元组如下：

```
[[It, is, a, day], nice]
[[It, is, a, day],brilliant]
```

而 Skip-Gram 的输入输出元组如下所示：

```
[It, nice], [is, nice][a,nice], [day, nice]
[It, brilliant], [is, brilliant], [a, brilliant], [day, brilliant]
```

我们希望模型能够理解 nice 和 brilliant 是两个有略微差异的单词（brilliant 意味着更好）。这些在含义上有细微差别的词称为 nuances（细微之处）。我们可以看到，对于 CBOW 来说，很有可能它会关注到 nice 和 brilliant 是同一个对象，因为它们的语义是通过周围的单词（It、is、a 和 day）得到的平均语义，所以 CBOW 模型给出的两个单词是一样的也就是很自然的事情了。对于 Skip-Gram，单词 nice 和 brilliant 是与 It、is 和 day 分开的，这样 Skip-Gram 更多地关注起单词（例如 brilliant 和 nice）之间的细微差异。

这里要注意，现实中模型可能有数百万个参数，要训练这些模型，需要大量数据。CBOW 以某种方式只是平均给定语境中所有单词的语义（例如，It is a day 的平均语义）；Skip-Gram 会学习更细致的单词表示，所以 Skip-Gram 将需要更多数据，一旦提供了足够多的数据，Skip-Gram 模型就很可能会优于 CBOW 模型。

4.6 词嵌入算法的扩展

Mikolov 等人在 2013 年发表的论文中讨论了几种可以进一步提高词嵌入学习算法性能的扩展，虽然它们最初被引入 Skip-Gram，但它们也可以扩展到 CBOW。此外，正如我们已经看到的，CBOW 在上面的示例中存在优于 Skip-Gram 算法的地方，本节将继续使用 CBOW 来解读词嵌入（Word Embedding）算法的扩展内容。

4.6.1 使用 Unigram 分布进行负采样

1. 相关理论概要

负采样的思想最初来源于一种叫作 Noise-Contrastive Estimation（NCE，噪音对比估计）的算法，原本是为了解决那些无法归一化的概率模型的参数预估问题。与改造模型输出概率的 Hierarchical Softmax 算法不同，NCE 算法改造的是模型的似然函数。

由上面的内容，我们知道，当通过从某些分布采样而不是从均匀分布进行采样时，负采样的性能会更好，Unigram 分布就是这样的分布。词w_i的一元概率由以下等式给出：

$$U(w_i) = \frac{count(w_i)}{\sum_{(j \in Corpus)} count\,(w_j)}$$

这里，$count(w_i)$是词w_i在文档中出现的次数。

让我们用一个例子来更好地理解 Unigram 分布。考虑以下句子：

```
Bob is a football fan. He is on the school football team.
```

在这里，单词“football”的 Unigram 概率如下：

$$U(\text{football}) = 2/26 = 1/13$$

这样我们可以看出，常用词的 Unigram 概率会更高，但是，在现实工作任务中，这些常用词往往是无效信息的词，例如 a 和 is 等。这样来看，在成本优化期间，这种高频率的词将被更多地负采样，导致含有更多信息性的词减少了被负采样的机会。因此，在优化期间，需要使用 Unigram 分布进行负采样在常用词和罕见词之间做好平衡，从而获得更好的性能。下面，我们来实现基于 Unigram 的负采样。

2. 实现基于 Unigram 的负采样

在这里，我们将看到如何使用 TensorFlow 实现基于 Unigram 的负采样：

```
unigrams = [0 for _ in range(vocabulary_size)]
for word,w_count in count:
    w_idx = dictionary[word]
```

```
        unigrams[w idx] = w count*1.0/token count
        word count dictionary[w idx] = w count
```

这里，count 是一个元组列表，其中每个元组由（word ID, frequency）组成。该算法计算每个词的 Unigram 概率，并将它们作为按词索引排序的列表返回。完整代码详见 ch4 文件夹中 4 comparison.ipynb 文件。

接下来，我们计算一个嵌入查找运算，就像我们通常为 CBOW 做的那样：

```
    train dataset = tf.placeholder(tf.int32, shape=[batch size,window_size*2])
    train labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
    # 相关变量.
    # embedding, vector for each word in the vocabulary
    embeddings = tf.Variable(tf.random_uniform([vocabulary_size,embedding_size],
-1.0, 1.0, dtype=tf.float32))
    softmax_weights =tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size], stddev=1.0 / math.sqrt(embedding_size), dtype=tf.float32))
    softmax_biases =tf.Variable(tf.zeros([vocabulary_size], dtype=tf.float32))
    stacked_embedings = None

    for i in range(2*window_size):
        embedding_i = tf.nn.embedding_lookup(embeddings,train_dataset[:,i])
        x_size,y_size = embedding_i.get_shape().as_list()
        if stacked_embedings is None:
            stacked_embedings =tf.reshape(embedding_i,[x_size,y_size,1])
        else:

            stacked_embedings =tf.concat(axis=2,values=[stacked_embedings,tf.
reshape(embedding_i,[x_size,y_size,1])])
    mean_embeddings = tf.reduce_mean(stacked_embedings,2,keepdims=False)
```

接下来，我们将基于 Unigram 分布对负样例进行采样。为此，我们将调用 TensorFlow 的内置函数 tf.nn.fixed_unigram_candidate_sampler：

```
    candidate_sampler = tf.nn.fixed_unigram_candidate_sampler(true_classes =
tf.cast(train_labels,dtype=tf.int64),num_true = 1, num_sampled =
num_sampled,unique = True,range_max = vocabulary_size,
                        distortion=0.75,num_reserved_ids=0, nigrams=unigrams,
name='unigram_sampler')
    loss = tf.reduce_mean(tf.nn.sampled_softmax_loss(weights=softmax_weights,
                                biases=softmax_biases, inputs=mean_embeddings,
                                labels=train_labels, num_sampled=num_sampled,
                                num classes vocabulary size,
    sampled values candidate sampler))
```

通常，我们的实现过程会执行以下步骤：

（1）定义变量、占位符和超参数。

（2）对于每批数据，会有以下情况：

- 通过查找上下文窗口每个索引的词向量并对它们求平均来计算平均输入词向量矩阵。
- 通过负采样计算损失，根据 Unigram 分布进行采样。
- 使用随机梯度下降优化神经网络。

4.6.2 降采样

1. 相关理论概要

降采样，也就是我们常说的子采样，也有下采样之说，其基本思想就是在模型训练时根据概率情况随机丢掉高频词（通常是一些无效词，例如停顿词等）。在数学上，这是通过忽略语料库中词序列中的词w_i来实现的。

$$p_{discard}(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

其中，t 是一个先验参数，一般取为$10^{(-5)}$，它用于控制被忽略词的词频率阈值。$f(w_i)$是w_i在语料中出现的频率。这有效地降低了停用词（例如，“the”“a”“of”“。”和“，”）的频率，从而在数据集中创建更多平衡。

相关实验研究表明，这种降采样技术可以显著提高低频词词向量的准确度。

2. 降采样实施

降采样的实现思路是从原始序列中创建一个新的词序列，通过从刚刚看到的概率中删除序列中的词，并使用这个新的词序列来学习词向量。完整代码见 ch4 文件夹中的 4_comparison.ipynb 文件。

4.6.3 CBOW 和其扩展类型比较

对于 CBOW、基于 unigram 负采样的 CBOW（Unigram）、基于 Unigram 负采样和降采样的 CBOW 的对比，通过代码执行得到图 4-29 所示的结果，我们从中可以看到随着迭代次数的增加，各个类型的损失函数的变化情况。完整代码见 ch4 文件夹中的 4_comparison.ipynb 文件。

通过图示，我们发现一个现象：基于 Unigram 负采样和降采样的 CBOW 与基于仅有 Unigram 负采样的 CBOW 相比，随着迭代次数的增加，损失函数的变化曲线基本一致。然而，实际上，这不应该被误解为降采样在学习问题的能力上缺乏优势。之所以有这种特殊情况出现，是因为与降采样一样，我们去掉了许多无用词（无信息词）引起文本质量随之提升（就信息质量而言）。然而，这也反过来导致学习问题的能力提升变得更加困难，因为在最初的问题设置中，词向量有机会在优化过程中利用大量无信息词，而在新的问题设置中，这种机会变得很少。



图 4-29 原始 CBOW 及其两个扩展类型的损失变化

4.7 结构化 Skip-Gram 和连续窗口模型

Word2vec 技术在获取词语义方面非常强大，但是它们并非没有限制。例如，它们不关注上下文词和目标词之间的距离，但是如果上下文词远离目标词，那么其对目标单词的影响会更小。因此，我们需要讨论在上下文中单独关注不同词所在位置的技术。Word2vec 的另一个限制是它在计算词向量时只关注给定词周围一个非常小的窗口，然而，实际上，应该考虑在整个语料库中共同出现的词的方式来计算合适的词向量。因此，我们将研究一种技术，它不仅可以查看词的上下文，还可以查看词的全局共现信息。

4.7.1 结构化 Skip-Gram 算法

先前讨论的 Skip-Gram 算法及其所有变体忽略了给定上下文中词的本地化。换句话说，标准 Skip-Gram 算法无法利用上下文词的准确位置，而只能同等地处理给定上下文中的所有词。例如，让我们考虑一个句子：

```
The dog barked at the mailman.
```

我们考虑目标词 barked，那么 barked 这个词的上下文就是“the”“dog”“at”和“the”“mailman”。我们将组成 5 个数据点（“barked”，“the”）、（“barked”，“dog”）、（“barked”，“at”）、（“barked”，“the”）和（“barked”，“mailman”），其中元组的第一个元素是输入词，第二个元素是输出词。如果我们考虑来自这个集合的两个数据点（“barked”，“the”）和（“barked”，“dog”），标准 Skip-Gram 算法将在优化过程中平等对待这两个元组。换句话说，Skip-Gram 忽略了上下文中词的实际位置，而从语言学的角度来看，元组（“barked”，“dog”）显然比（“barked”，“the”）包含更多的信息。接下来，结构化的 Skip-Gram 算法将试图解决这个限制。让我们具体看一下。

如图 4-30 所示，结构化的 Skip-Gram 算法使用下面的架构来解决上面标准 Skip-Gram 算法遇到的限制问题。

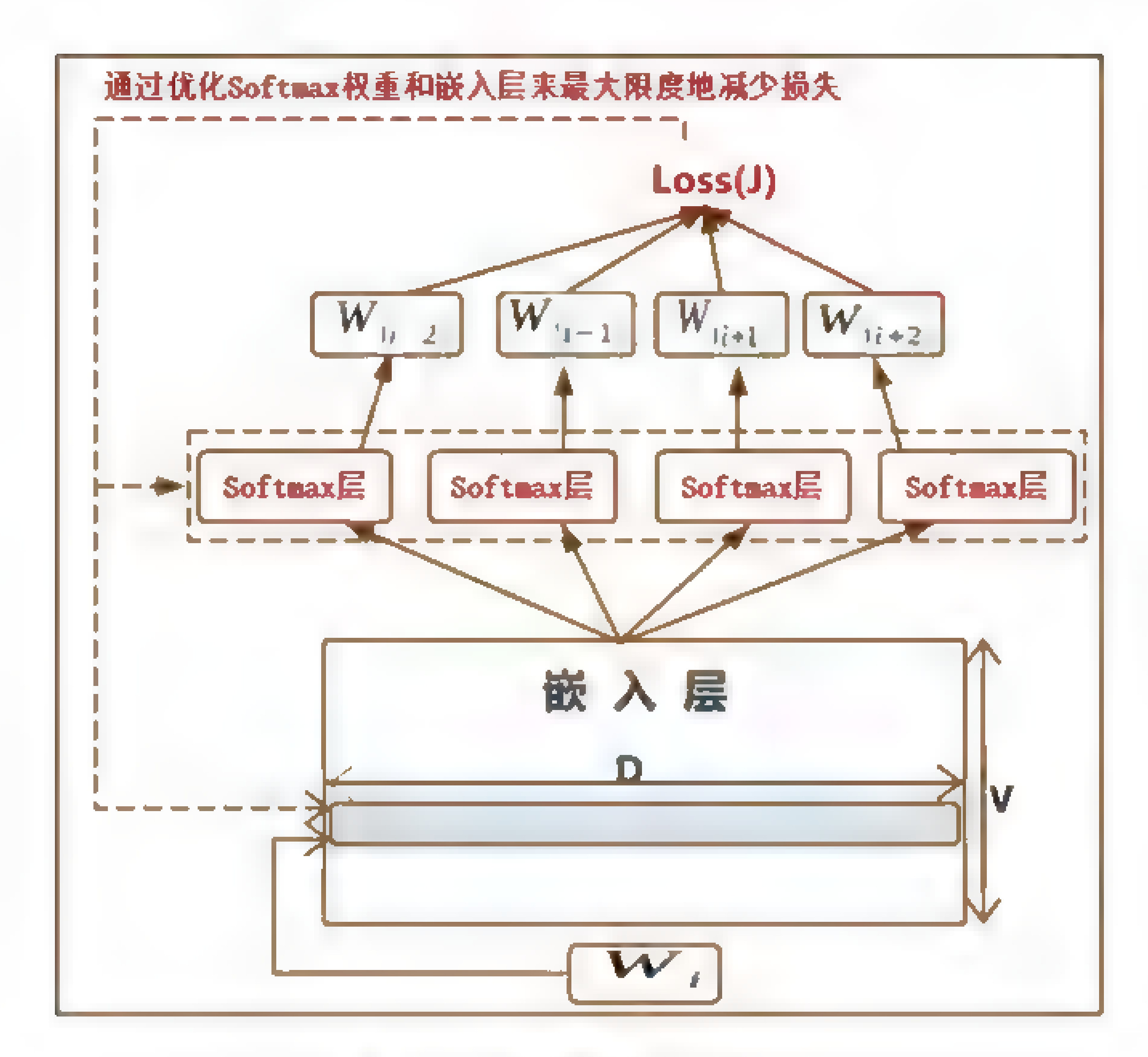


图 4-30 结构化的 Skip-Gram

如图 4-30 所示，结构化 Skip-Gram 在优化期间保留了上下文词的结构或本地化的形式，但是它需要更多的内存空间，因为参数的数量与窗口大小呈线性关系。具体来讲，假设 Skip-Gram 模型的窗口大小为 m（目标词一侧的词数量），如果标准 Skip-Gram 模型在 softmax 层有 P 个参数，那么结构化 Skip-Gram 模型将有 $2mP$ 个参数，因为我们在上下文窗口中为每个位置都设置了一组 P 参数。

关于损失函数，标准 Skip-Gram 模型的负采样 softmax 损失函数如下所示：

$$J(\theta) = -\left(\frac{1}{(N-2m)}\right)\sum_{i=m+1}^{N-m}\sum_{j\neq i j=i-m}^{i+m} log\left(\sigma\left(logit(x_n)_{(w_j)}\right)\right) + \sum_{q=1}^{k} E_{(w_q-vocabulary-w_i,w_j)}\, log\left(\sigma\left(-logit_p(x_n)_{(w_q)}\right)\right)$$

而结构化 Skip-Gram 使用以下损失函数：

$$J(\theta) = \sum_{p=1}^{2m} -\left(\frac{1}{(N-2m)}\right)\sum_{i=m+1}^{N-m}\sum_{j\neq i j=i-m}^{i+m} log\left(\sigma\left(logit_k(x_n)_{(w_j)}\right)\right) + \sum_{q=1}^{k} E_{(w_q-vocabulary-w_i,w_j)}\, log\left(\sigma\left(-logit_p(x_n)_{(w_q)}\right)\right)$$

这里，使用第 p 组 softmax 权重值和对应于w_j位置索引的 softmax 偏差来计算$\mathrm{logit}_p(x_n)_{(w_j)}$。

完整实现代码，可参考 ch4 文件夹中的 4_word2vec_extended.ipynb 文件。

首先，我们将定义输入和输出占位符：

```
    train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
    train_labels = [tf.placeholder(tf.int32, shape=[batch_size, 1]) for _ in
range(2*window_size)]
```

然后我们将从训练输入和标签处开始定义计算损失：

```
    # 相关变量
    embeddings = tf.Variable(
                    tf.random_uniform([vocabulary_size, embedding_size],-1.0, 1.0))
    softmax_weights = [tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size],stddev=0.5 / math.sqrt(embedding_size))) for _ in
range(2*window_size)]
    softmax_biases =[tf.Variable(tf.random_uniform([vocabulary_size],0.0,0.01))
for _ in range(2*window_size)]
    # 相关模型

    embed = tf.nn.embedding_lookup(embeddings, train_dataset)
    # Compute the softmax loss, using a sample of the negative labels each time.
    loss = tf.reduce_sum(
    [

tf.reduce_mean(tf.nn.sampled_softmax_loss(weights=softmax_weights[wi],
    biases=softmax_biases[wi], inputs=embed, labels=train_labels[wi],
    num_sampled=num_sampled, num_classes=vocabulary_size))
                    for wi in range(window_size*2)
    ]
    )
```

结构化 Skip-Gram 解决了标准 Skip-Gram 算法的一个重要限制，即在学习过程中注意上下文词的位置。这是通过引入一组单独的 softmax 权重值和对上下文每个词位置的偏差来实现的，这将有助于优化模型性能，但也因为参数的增加而需求更多的内存空间。接下来，我们将看到与 CBOW 模型类似的扩展。

4.7.2 连续窗口模型

连续窗口模型（Continuous Window Model）是类似于结构化 Skip-Gram 模型而扩展出的 CBOW 模型。在原始 CBOW 算法中，在传给 softmax 层之前，将所有上下文词找到的词向量进行平均化处理，而在连续窗口模型中，不是对词向量进行平均，而是将它们连接起来，从而产生 $m \times D$ 长的词向量，其中 D_{emb} 是 CBOW 算法的原始词向量大小（Embedding Size）。图 4-31 给出了连续窗口模型结构。

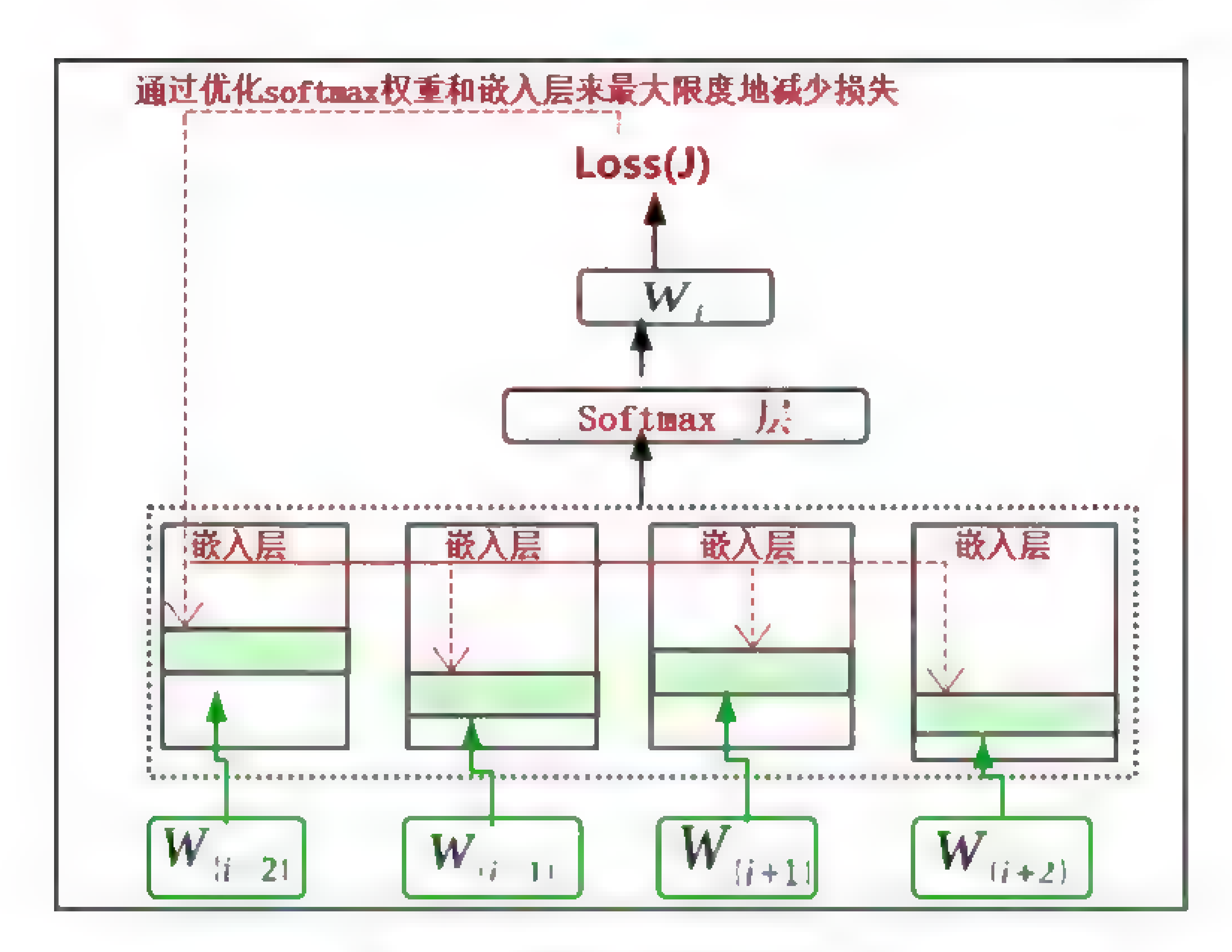


图 4-31 连续窗口模型结构示意图

在本节中，我们讨论了两种扩展的 Skip-Gram 和 CBOW 算法，这两种变体基本上利用了上下文中词的位置，而不是同等地看待给定上下文中的所有词。接下来，我们将讨论另一种著名模型——GloVe(Global Vectors Representation，全局向量表示)模型，我们将看到GloVe克服了Skip-Gram 和 CBOW 固有的某些限制。

4.8 GloVe 模型

关于学习词向量表示，这里主要有两类模型系列：①全局矩阵分解方法，如潜在语义分析(LSA)(Deerwester 等，1990)；②局部上下文窗口方法，如 Skip-Gram 和 CBOW 模型(Mikolov 等，2013)。但是，目前这两类模型都有明显的缺点，例如，尽管 LSA 这样的方法能有效地利用统计信息，但它们在词类比任务上的表现相对较差，这表明了它们次优的向量空间结构。Skip-Gram 这样的方法可能在词类比上表现更好，但它们在利用语料库的统计信息上表现并不好，因为它们是在分离的局部上下文窗口中训练的，而不是在全局的共现（Co-Occurrence）计数上训练的。GloVe 模型试图充分整合利用这两个领域——一种有效利用全局语料库统计数据的方法，同时以基于上下文窗口的方式优化学习模型，类似于 Skip-Gram 或 CBOW。

对于 GloVe 模型而言，其主要目标是将词进行向量化表示，以便使各个向量之间能够尽可能多地涵盖语境内的语义和语法信息。通过输入语料库而输出词向量。实现方法为：首先基于整个语料库构建词的共现矩阵，然后基于共现矩阵和 GloVe 模型处理学习词向量。

滑铁卢大学 Vineet John 于 2017 年 4 月撰写的一篇论文《*A Survey of Neural Network Techniques for Feature Extraction from Text*》中提出，任意词之间的关系都可以通过研究它们的共现概率与多

个探测词（Probe Word）之间的比例来检验，且词向量学习的合理起点应该是共现概率的比例，而非概率本身。那么我们可以将这种共现关系表示成以下形式：

$$F((w_i - w_j)^T w_k) = \frac{P_{(ik)}}{P_{(jk)}}$$

4.8.1 共现矩阵

设共现矩阵为 X，其元素为$X_{(i,j)}$。

这里，$X_{(i,j)}$是指在整个语料库中词 i 和词 j 共同出现在一个窗口中的次数。

下面以一句话为例进行解读：

```
The dog barked at the mailman.
```

上面这个句子就是一个语料库，该语料库包含 5 个单词：the、dog、barked、at、mailman。

定义单侧窗口大小 window_size 为 2，则整个窗口宽度（大小）为 2*window_size+ 1 =5，那么我们就可以给出窗口宽度为 5 的统计窗口，具体如表 4-1 所示。

表 4-1 窗口宽度为 5 的统计窗口

窗口标号	中心词	窗口内容
0	the	the dog barked
1	dog	the dog barked at
2	barked	the dog barked at the
3	at	dog barked at the mailman
4	the	barked at the mailman
5	mailman	at the mailman

考虑到中心词两侧保证至少 2 个单词的情况，现在以中心词为 at 的窗口内容为例，语境词为 dog barked at the mailman，则将整个窗口遍历一次即可得到共现矩阵 X，如下：

$$X_{(at,dog)} \mathrel{+}= 1$$
$$X_{(at,barked)} \mathrel{+}= 1$$
$$X_{(at,the)} \mathrel{+}= 1$$
$$X_{(at,mailman)} \mathrel{+}= 1$$

4.8.2 使用 GloVe 模型训练词向量

我们假设 i="dog"和 j="cat"，且给出一个探测词 k，那么可以定义P_{ik}为词 i 和词 k 在一起出现

的概率，P_{jk}为词 j 和词 k 一起出现的概率。这时，对于 k="bark"，它很可能与 i 一起出现，因此P_{ik}会很高；然而，k 不会经常与 j 一起出现则导致低P_{jk}。因此，下式成立：

$$P_{ik}/P_{jk} \gg 1$$

如果 k="whiz"，那它不太可能出现在 i 的附近，因此将具有低的P_{ik}；但由于 k 与 j 高度相关，因此P_{jk}的值将很高。所以，这会导致以下结果：

$$P_{ik}\ /P_{jk} \approx 0$$

如果 k="pet"，那它与 i 和 j 都有很强的关系，或者 k="lawyer"，其中 i 和 j 两者都有最小的相关性，我们可以得到：

$$P_{ik}\ /P_{jk} \approx 1$$

由此我们得知，通过统计彼此相近两个词的频数可以计算其对应共现概率 P_{ik}、P_{jk}，进而可以得到二者比率与 1 的关系，最终可以得到词之间的关系情况。所以，P_{ik} / P_{jk}就成为学习词向量的重要方法，下面给出最通用的表示形式：

$$F((w_i - w_j)^T w_k) = \frac{P_{(ik)}}{P_{(jk)}}$$

经过一系列的推导，我们最终会得到以下损失函数：

$$J = \sum_{i,j=1}^{V} f(X_{ij})\,(w_i^T \widetilde{w_j} + b_i + \widetilde{b_j} - logX_{ij})^2$$

关于这里的推导过程，感兴趣的读者可以查看 Jeffrey 等人的论文《*GloVe: Global Vectors for Word Representation*》，里面给出了详细的推导过程。下面我们看看 GloVe 模型的实现。

4.8.3　GloVe 模型实现

这部分的完整代码见 ch4 文件夹中的 4_glove.ipynb 文件。

首先，我们将定义输入和输出：

```
    train_dataset = tf.placeholder(tf.int32,
shape=[batch_size],name='train_dataset')
    train_labels = tf.placeholder(tf.int32,
shape=[batch_size],name='train_labels')
```

接下来，我们将定义两个不同的词嵌入层：一个用于查找输入词，另一个用于查找输出词。另外，我们将定义词向量偏差，就像我们对 softmax 层的偏差一样：

```
    in_embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
embedding_size], -1.0, 1.0), name='embeddings')
    in_bias_embeddings = tf.Variable(tf.random_uniform([vocabulary_size],
```

```
0.0,0.01, dtype=tf.float32), name='embeddings_bias')
    out_embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
embedding_size], -1.0, 1.0), name='embeddings')
    out_bias_embeddings = tf.Variable(tf.random_uniform([vocabulary_size],
    0.0,0.01, dtype=tf.float32), name='embeddings_bias')
```

现在，我们将查找给定输入词和输出词（标签）对应的词向量：

```
embed_in = tf.nn.embedding_lookup(in_embeddings, train_dataset)
embed_out = tf.nn.embedding_lookup(out_embeddings, train_labels)
embed_bias_in = tf.nn.embedding_lookup(in_bias_embeddings, train_dataset)
embed_bias_out = tf.nn.embedding_lookup(out_bias_embeddings, train_labels)
```

此外，我们将在损失函数中定义 $f(X_{ij})$（weights_x）和 X_{ij}（x_ij）的占位符：

```
weights_x = tf.placeholder(tf.float32, shape=[batch_size],name='weights_x')
x_ij = tf.placeholder(tf.float32, shape=[batch_size], name='x_ij')
```

最后，我们将定义完全损失函数，如下所示：

```
loss = tf.reduce_mean(weights_x * (tf.reduce_sum(embed_in*embed_out,axis=1) +
                      embed_bias_in + embed_bias_out - tf.log(epsilon+x_ij))**2)
```

在本节中，我们研究了词嵌入的另一个重要技术：GloVe 模型。GloVe 相对于之前的 Word2vec 技术，主要优点是它注重语料库的全局和局部统计学习，以便更深入地学习词向量。由于 GloVe 能够捕获有关词的全局信息，因此它们往往会提供更好的性能表现，尤其是当语料库大小增加时。另外，与 Word2vec 技术不同，GloVe 不是近似损失函数（例如，使用负采样的 Word2vec），而是计算真实损失函数，这样对于模型损失函数的优化会更加有利。

4.9 使用 Word2Vec 进行文档分类

尽管 Word2vec 提供了一种非常优雅的方法来学习词的数字表示，正如我们在定量上看到的（损失值）和定性上看到的（t-SNE 词嵌入），仅仅进行学习词表示并不足以说明在现实世界中词向量的应用能力。其实，词嵌入（Word Embedding）已被用于许多工作任务的词特征表示中，例如图像标题生成和机器翻译。然而，这些任务涉及不同学习模型的组合（例如卷积神经网络（CNN）和长短期记忆（LSTM）模型或两个 LSTM 模型）。这些将在后面的章节中讨论。为了理解词嵌入的实际使用方法，这里将使用一个较为简单的工作任务——文档分类。

文档分类是 NLP 中最受欢迎的任务之一。文档分类对于处理大量数据集合的人员非常有用，例如新闻网站、出版商和大学。因此，有意思的是学习词向量如何通过嵌入整个文档而不是词来适应真实世界的任务，比如说文档分类便是如此。

本节的代码见 ch4 文件夹中的 4_document_embedding.ipynb 文件。

4.9.1 数据集

对于此任务，我们将使用已组织好的一组文本文件，这些都是 BBC 的新闻文章，本系列中的每个文档都属于以下类别之一：商业、娱乐、政治、体育或技术。我们使用每个类别中的 250 个文档，词汇量大小为 25000 个。此外，每个文档将由<文档类型>-<id>标记表示，以便进行可视化。例如，娱乐部分的第 50 个文档将表示为 entertainment-50。应该注意的是，与在实际应用中分析的大型文本语料库相比，这是一个非常小的数据集。然而，目前这个小例子足以看到词向量的性能表现情况。

以下是来自实际数据的几个简短片段：

```
Business
Japan narrowly escapes recession
Japan's economy teetered on the brink of a technical recession in the three months to September, figures show.
Revised figures indicated growth of just 0.1% - and a similar-sized contraction in the previous quarter. On an annual basis, the data suggests annual growth of just 0.2%,...
Technology
UK net users leading TV downloads
British TV viewers lead the trend of illegally downloading US shows from the net, according to research.
New episodes of 24, Desperate Housewives and Six Feet Under, appear on the web hours after they are shown in the US, said a report. Web tracking company Envisional said 18% of downloaders were from within the UK and that downloads of TV programmers
had increased by 150% in the last year.…
```

4.9.2 使用词向量对文档进行分类

这里，我们看看诸如 Skip-Gram 或 CBOW 之类的词嵌入方法是否可以扩展到文档分类或文档聚类应用中。由于 CBOW 算法已被证明在使用更小的数据集上比 Skip-Gram 表现更好，因此，这里我们将使用 CBOW 算法。

（1）从所有文本文件中提取数据并学习词向量。

（2）从已经训练过的文档中提取一组随机文档。

（3）扩展学习词向量以嵌入这些选定的文档。更具体地说，我们将通过找到属于文档中的所有词向量的平均值来表示文档。

（4）使用 t-SNE 可视化技术找到文档向量，以查看词向量是否可用于文档聚类或分类。

（5）最后，可以使用诸如 K-means 之类的聚类算法来为每个文档分配标签。

1. 学习词向量

首先，我们将为训练数据、训练标签、验证数据（用于监控词向量）和测试数据（用于计算

测试文档的平均向量）定义多个占位符：

```
    # Input data.
    train_dataset = tf.placeholder(tf.int32,shape=[batch_size, 2*window_size])
    train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
    test_labels = tf.placeholder(tf.int32,shape=[batch_size],
name='test_dataset')
```

接下来，我们将为词汇表、softmax 权重值和偏差定义词向量变量（用于计算测试文档的平均向量）：

```
    embeddings = tf.Variable(tf.random_uniform([vocabulary_size,embedding_size],
-1.0, 1.0, dtype=tf.float32))
    softmax_weights = tf.Variable(
      tf.truncated_normal([vocabulary_size, embedding_size],
      stddev=1.0 / math.sqrt(embedding_size), dtype=tf.float32))

    softmax_biases = tf.Variable(tf.zeros([vocabulary_size], dtype=tf.float32))
```

然后定义采样的负 softmax 损失函数：

```
    loss = tf.reduce_mean(tf.nn.sampled_softmax_loss(weights=softmax_weights,
      biases=softmax_biases, inputs=mean_embeddings,
      labels=train_labels, num_sampled=num_sampled,num_classes=vocabulary_size))
```

2. 词嵌入到文档嵌入

为了从词嵌入中获得良好的文档嵌入，我们将一个文档中所有词向量平均化作为该文档向量。这里我们将分批处理相关数据。为此，我们需要完成以下内容来达到此目的。

对于每个文档，执行以下操作：

（1）创建数据集，其中每个数据点都是属于文档的词。

（2）对于从数据集中采样到的小批量，通过对小批量中所有词向量求平均来返回平均向量。

（3）批量遍历测试文档并通过平均小批量向量来获取文档向量。

我们将得到的平均批量向量如下：

```
    mean_batch_embedding = tf.reduce_mean(tf.nn.embedding_lookup(embeddings,
test_labels), axis=0)
    mean_embeddings = tf.reduce_mean(stacked_embeddings, 2,keepdims=False)
```

接着，我们将在文档中所有批次的列表中收集这些平均向量，并将这些平均向量作为文档向量。这是获取文档向量的一种非常简单却强大的方法，下面就会看到。

3. 文档向量的文档聚类和 t-SNE 可视化

在图 4-32 中，我们对 CBOW 算法学习的文档向量进行了可视化，可以看到该算法在学习具有同样主题的文档方面上表现还不错。正如我们之前讨论的那样，在以无人监督的方式对文档进行分类/聚类方面，这种简单的方法已被证明是一种非常有效的方法。

4. 检查异常值

从图 4-32 可以看出，产生异常值的文档貌似不多，sport-130 和 sport-50 就属于少见的情况，下面我们对于这些文档的内容进行查看分析，以便查找产生这种现象的原因。

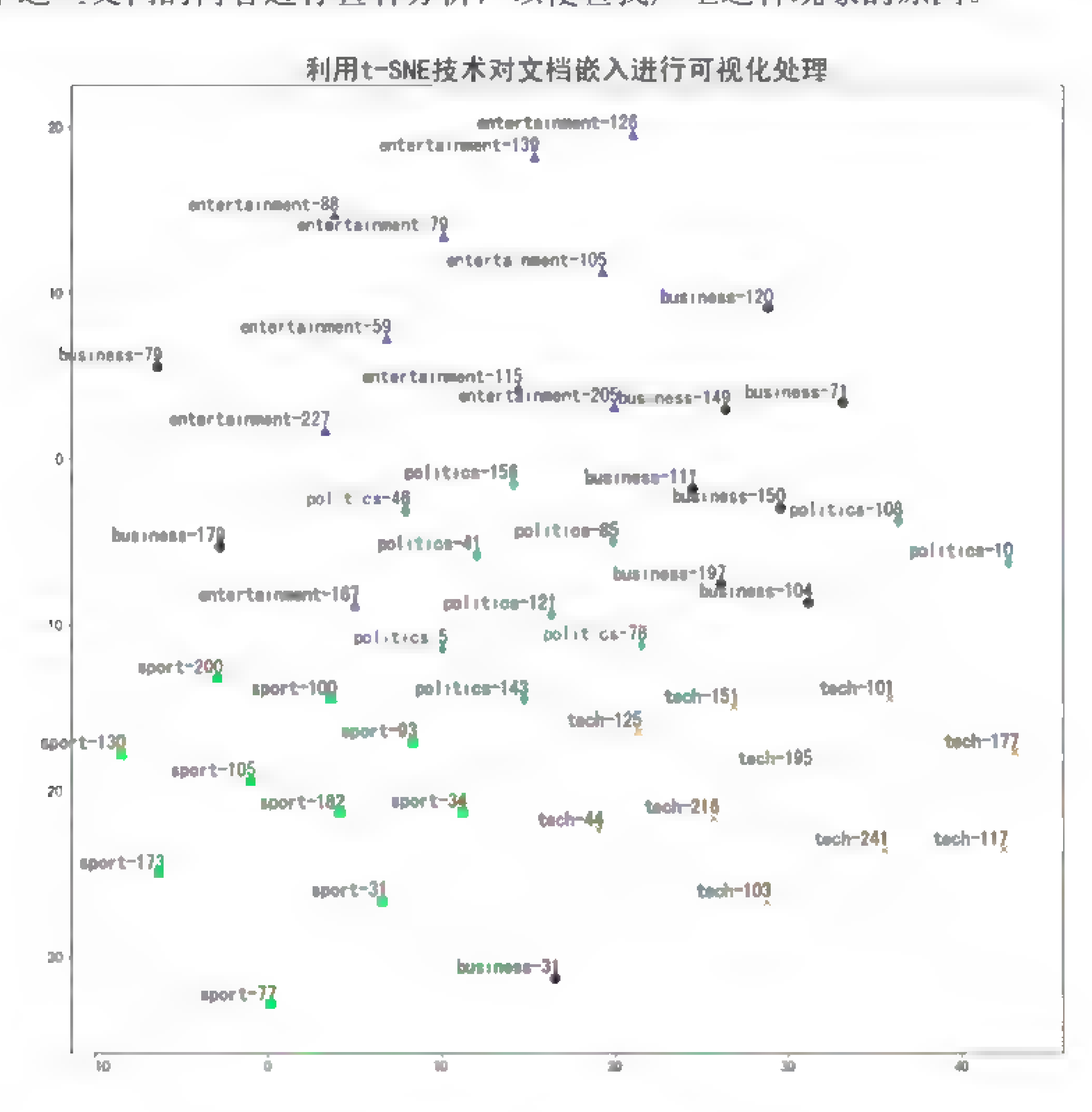


图 4-32 文档向量的文档聚类及其 t-SNE 可视化

以下是 sport-130 文档的片段：

```
sport-130

Weir poised for Sunderland move
Larne's teenage star Robbie Weir is poised to join Sunderland after turning down
a move to Stoke City.
The 17-year-old Irish League midfielder was also being chased by Rangers and
Fulham, but Mick McCarthy's side appear to have won the race. But Larne boss Jimmy
McGeough has yet to confirm that Weir is on his way from Inver Park. "I heard on
```

```
Sunday that he has joined Sunderland, but not from the lad himself," he said. ''Robbie
has an agreement with Larne that he can negotiate with interested clubs.
...
```

本文档主要讨论爱尔兰联盟中场球员罗比威尔加盟桑德兰事件的影响，而不是像其他 Sport 文档里面在讨论具体的体育比赛的。

以下是 sport-50 文档的片段：

```
    IAAF awaits Greek pair's response
    Kostas Kenteris and Katerina Thanou are yet to respond to doping charges from
the International Association of Athletics Federations (IAAF).
    The Greek pair were charged after missing a series of routine drugs tests in
Tel Aviv, Chicago and Athens. They have until midnight on 16 December and an IAAF
spokesman said: "We're sure their responses are on their way." If they do not respond
or their explanations are rejected, they will be provisionally banned from
competition. They will then face a hearing in front of the Greek Federation,...
```

我们可以阐明为什么 sport-50 远离其他与体育相关的文章聚集在一起。让我们仔细看看另一个接近 sport-50 的文档，即 Entertainment-115：

```
    Entertainment-115
    Rapper Snoop Dogg sued for 'rape'
    US rapper Snoop Dogg has been sued for $25m (£13m) by a make-up artist who
    claimed he and his entourage drugged and raped her two years ago.
    The woman said she was assaulted after a recording of the Jimmy Kimmel Live TV
    show on the ABC network in 2003. The rapper's spokesman said the allegations
    were "untrue" and the woman was "misusing the legal system as a means of
    extracting financial gain". ABC said the claims had "no merit". The star has
    not been charged by police.
```

因此，该地区的文件似乎与各种犯罪或非法指控有关，而不是与体育或娱乐有关。这使得文档远离其他典型的体育或娱乐相关文档进行聚类。

5. 使用 K-means 对文件进行聚类/分类

到目前为止，我们已经能够直观地检查文档情况，但是这还不够，因为如果有 1000 多个文档需要聚类/分类，就必须在视觉上检查 1000 次，所以我们需要更自动化的方法来实现这一目标。

我们可以使用 K-means 来聚类这些文档。K-means 是一种简单但功能强大的技术，用于根据数据的相似性将数据分成组（集群），因此类似的数据将位于同一组中，不同的数据将位于不同的组中。K-means 的工作方式如下：

（1）定义 K，即要形成的簇的数量。我们将其设置为 5，表示有 5 个类别。

（2）形成 K 个随机质心，它们是簇的中心。

（3）将每个数据点分配给最近的聚类质心。

（4）将所有数据点分配给某个集群后，我们将重新计算集群质心（数据点的平均值）。

（5）以这种方式继续，直到质心运动变得小于某个阈值。

这里使用 scikit-learn 库来获得 K-means 算法。在代码中，如下所示：

```
kmeans = KMeans(n_clusters=5, random_state=43643, max_iter=10000,n_init=100,
algorithm='elkan')
```

最重要的超参数是 n_clusters，它是我们想要形成的聚类数。可以使用其他超参数来查看它们对性能的影响。有关超参数的解释，请访问：http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html。

接着，将用于训练的文档分类到具体类中。我们将获得表 4-2 中的内容。

它看起来似乎并不完美，但是可以很好地将属于不同类别的文档分类到不同的标签。

表 4-2　获得的内容

类标签	文档
0	['business-179', 'entertainment-167', 'politics-5', 'sport-200', 'sport-93', 'sport-105', 'sport-31', 'sport-173', 'sport-130', 'sport-34', 'sport-182', 'sport-100']
1	['entertainment-79', 'entertainment-88', 'entertainment-105', 'entertainment-59', 'entertainment-139', 'entertainment-227', 'entertainment-126']
2	['politics-10', 'politics-78', 'tech-177', 'tech-117', 'tech-101', 'tech-125', 'tech-44', 'tech-103', 'tech-216', 'tech-195', 'tech-241', 'tech-151']
3	['business-71', 'business-111', 'business-104', 'business-31', 'business-150', 'business-149', 'business-197', 'business-120', 'business-79', 'entertainment-205', 'entertainment-115', 'politics-143', 'politics-121', 'politics-41', 'politics-85', 'politics-46', 'politics-108', 'politics-156']
4	['sport-77']

4.9.3　小结

在本节中，我们学习了如何将词嵌入扩展到分类/聚类文档。首先，学习了词嵌入。然后，我们通过对该文档中找到的所有词向量进行平均来创建文档向量。接着，我们使用文档向量来分类/聚类属于这些类别的 BBC 新闻文章：娱乐、科技、政治、商业和体育。在对文档进行聚类之后，我们看到文档被合理地聚类，使得属于一个类别的文档聚集在一起。但是，对于一些异常文件，我们在分析了这些文档的文本内容之后，发现这些文档背后存在某些合理的关联。

4.10 总结

在本章中，我们研究了 Word Embedding 的几种重要模型：Word2vec（Skip-Gram、CBOW）和 GloVe 模型，对原始 Skip-Gram、改进型 Skip-Gram、原始 CBOW、改进型 CBOW 等之间的性能差异做了对比分析。为了方便比较，我们使用了一种流行的二维可视化技术 t-SNE。

然后，我们又介绍了 Word2vec 算法的几个扩展，以提高其性能，然后是几个基于 Skip-Gram 和 CBOW 算法的新算法。结构化 Skip-Gram 通过在优化期间保留上下文词的位置来扩展 Skip-Gram 算法，允许算法基于它们之间的距离来处理输入词-输出词。相同的扩展也可以应用于 CBOW 算法，这里给出了连续窗口算法。

接下来，我们讨论了另一个词嵌入学习技术：GloVe 模型。GloVe 通过将全局统计数据合并到优化中，使当前的 Word2vec 算法更进一步，从而提高了性能。

最后，我们讨论了使用词向量-文档聚类/分类的实际应用，发现词向量非常强大，并且允许我们将相关文档合理地聚集在一起。

在下一章中，我们将讨论深度学习的代表算法之一——卷积神经网络（CNN），并将讲述如何使用 CNN 来利用句子的空间结构对它们分类。

第5章

卷积神经网络与句子分类

在本章中，我们将介绍一个众所周知的神经网络——卷积神经网络（CNN），根据维基百科的定义，它是一类深度前馈人工神经网络，最初是为解决图像识别等问题而设计的，以便降低对图像数据预处理的要求及避免复杂的特征提取。CNN 模型对缩放、平移、旋转等畸变具有不变性，有很强的泛化能力，这一点与 SIFT 等算法类似，所以它也被称为位移不变或空间不变的人工神经网络（SIANN）。而 CNN 最大的特点是卷积的权重值共享结构，可以大幅度减少神经网络的参数数量，防止过拟合，同时又降低了神经网络模型的复杂程度，这样一来，我们在进行深度模型训练时性能表现更佳且又不必担心内存溢出。虽然 CNN 最初主要是为解决图像问题而设计的，但是现在它在目标检测、视频识别、推荐系统和自然语言处理方面也得到广泛的应用。

在本章中，我们会对 CNN 的来龙去脉、组成部分、基本运算单元、基本原理进行详细解读，接着，我们会分析 4 类常见的经典卷积网络的结架构和特性，最后我们会给出两个应用案例：手写数字识别和基于卷积神经网络的句子分类。

5.1 认识卷积神经网络

5.1.1 卷积神经网络的历史演变

卷积神经网络的理念起源于早期科学家 Sherrington 首次提出的感受野概念（Receptive Field，《*Observations on the scratch - reflex in the spinal dog*》，第一次出版日期为 1906 年 3 月 13 日）。Sherrington(1906)首次使用术语“感受野”来描述可以在狗身体上引起划痕反射的皮肤区域。Hartline 于 1938 年将该术语应用于单个神经元。二十世纪五六十年代，Hubel 和 Wiesel 对皮质生理学使用了感受野组织这个概念，提出了这样的理论：在视觉系统一个层面上的细胞的感受野是由视觉系统较低层的细胞输入形成的。通过这种方式，可以组合小而简单的感受野，再形成大而复杂的感受野。后来的研究人员通过允许视觉系统的一个层次的细胞受到来自更高层次的反馈的影响，阐述了这种

简单的分层排列（来自维基百科）。二十世纪八十年代，日本科学家 Kunihiko Fukushima 提出了神经感知机（Neocognitron）的概念，被认为是卷积网络实现的最初原型。神经感知机中包含两类神经元，即 S-cell 和 C-cell。这里 S-cells 用于特征提取，对应现在主流卷积神经网络中的滤波（也就是卷积核）操作；C-cell 用于抗形变，对应现在的激活函数、最大池化等操作。

正常情况下，一个卷积神经网络由多个卷积层组成，而每个卷积层又会进行一系列操作，具体如下：

（1）通过多个不同的滤波和偏差提取出图像的局部特征，这样每一个卷积核都会映射出一个新的二维图像。

（2）对于（1）中卷积核滤波后的输出结果进行非线性激活函数处理。目前，ReLU 函数是最流行的非线性激活函数。

（3）对于（2）中激活函数后的结果再进行池化操作（降采样）。现在多数使用最大池化操作保留最显著的特征，并提升模型的畸变容忍能力。

卷积神经网络的发展路径如图 5-1 所示。关于卷积神经网络的历史沿革路径，这里就不做过多的解读了。

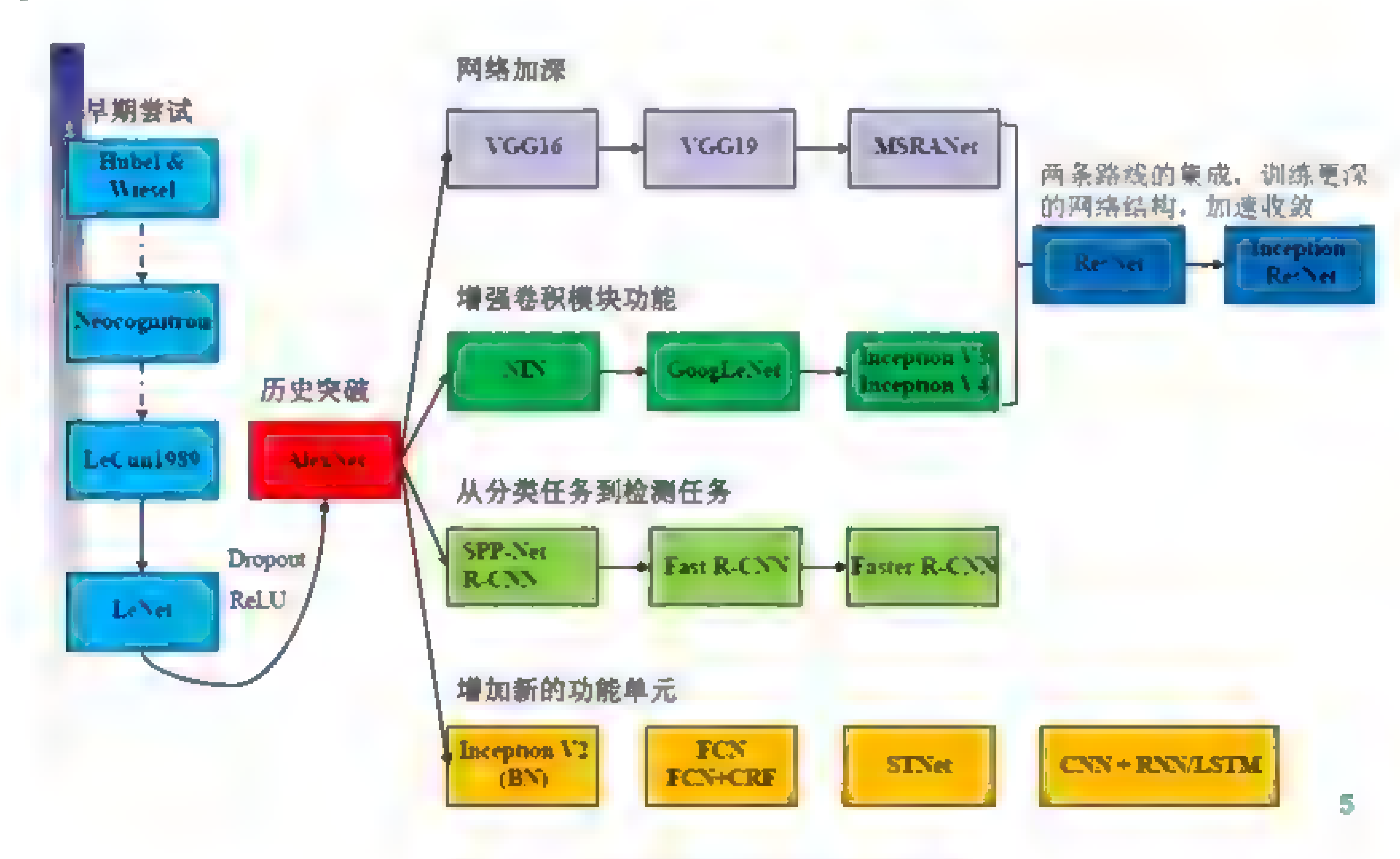


图 5-1　卷积神经网络发展路径图

5.1.2　卷积神经网络结构简述

最初，输入内容被连接到一组卷积层，这些卷积层在输入层上滑动一组权重值（有时也称为卷积窗口或过滤器），并通过卷积操作的方式产生一个输出。卷积层使用组织起来的少量权重值仅能覆盖每层中的一小部分输入，这一点不像全连接那样可以使得这些权重值在某些维度（例如，图像的宽度和高度尺寸）可以实现共享。由于 CNN 使用卷积运算，通过在符合要求的维度上滑动这个小的权重值集合来共享权重值以便产生输出，最终可以得到卷积运算的结果，如图 5-2 所示。如果

一个卷积过滤器输出的图案出现在原图像中，那么卷积输出的是高区位值，反之，就是低区位值. 后面会给出数字化解释。当然，我们也可以发现，通过卷积各运算之后，最后得到了一个矩阵，该矩阵可以展现出前面映射过来的图案是否在指定图像中出现，如图 5-2 所示。

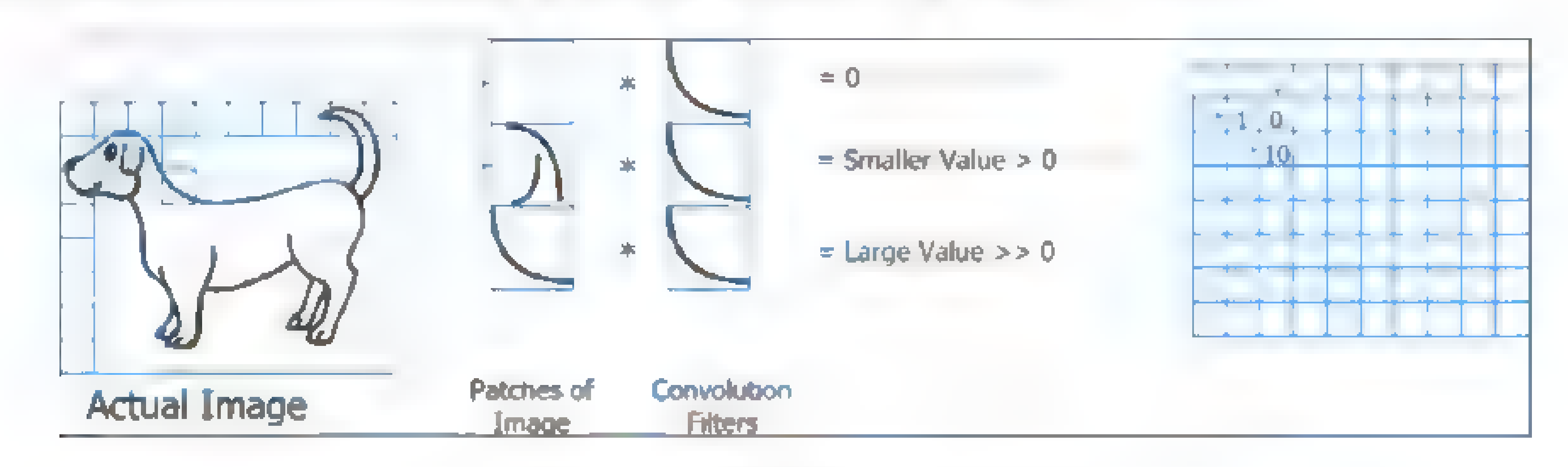


图 5-2 卷积操作在图像上的展示

此外，这些卷积层可选择与池化/子采样层混合在一起使用，这样就降低了输入的维度。当我们进行降维操作时，不但能迫使 CNN 学习较少的信息，而且可以让 CNN 的不变量进行转化，从而使得模型得到更好的归一化和正则化。通过将输入（例如图像）分成许多很小的小块并将每个小块转换成单个元素，维度就被降低了。在图 5-3 中，我们将说明如何进行池化运算使 CNN 的不变量进行平移。

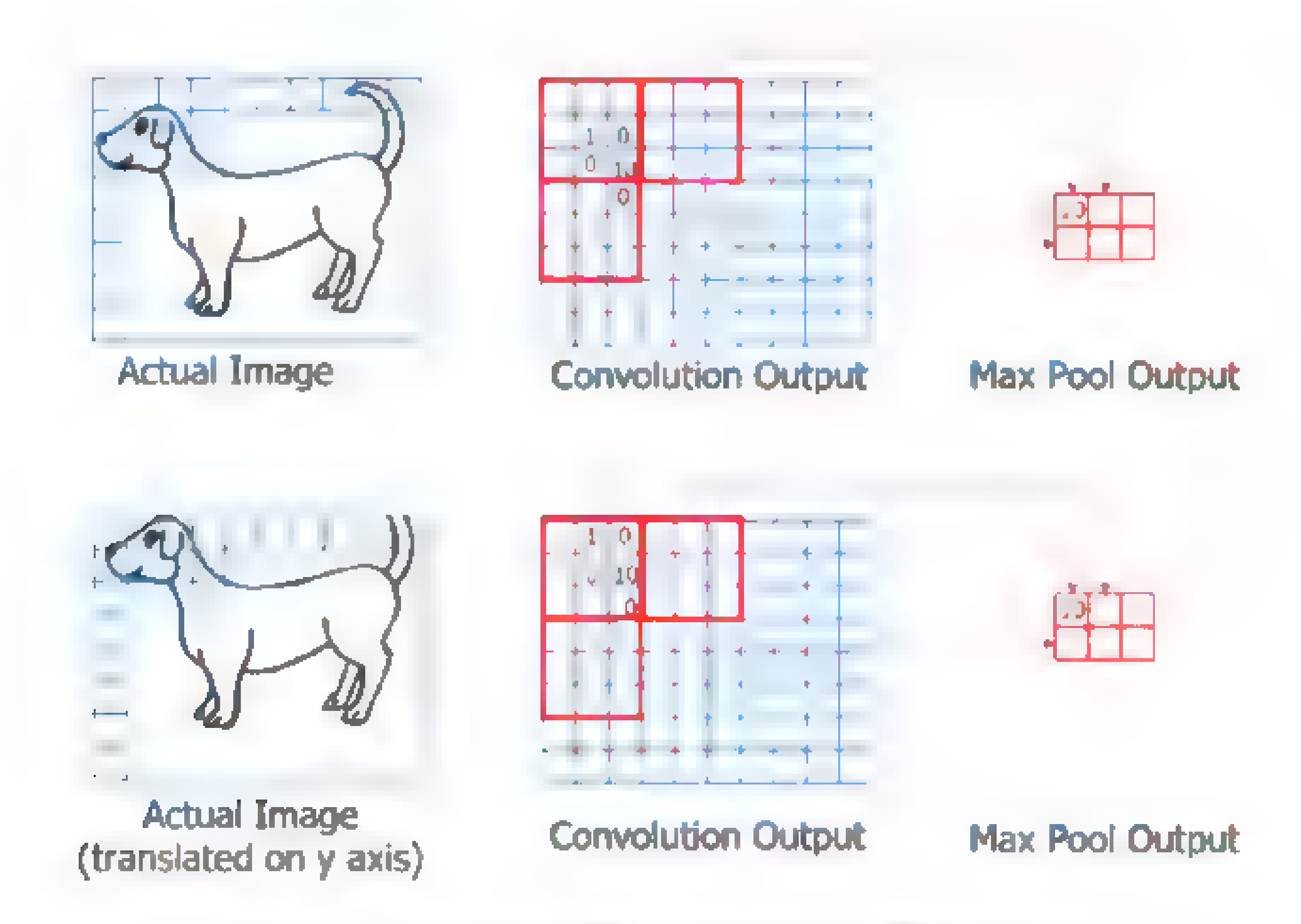


图 5-3 池化操作帮助 CNN 进行不变量平移

在图 5-3 中，我们有原始图像和在 y 轴上略微平移的图像。这两个图像通过卷积运算输出，可以看到值 10 出现在卷积输出中稍微不同的位置，但是使用最大池化（取每个粗线方块内的最大值）后，我们可以在最后获得相同的输出。稍后将详细讨论这些操作。

最后，上面的输出被回馈到一组全连接层，然后将全连接层后的输出转发到最终的分类/回归层（例如，句子/图像分类）。正常情况下，全连接层包含了 CNN 权重值总数的大部分，而卷积层只有一小部分权重值。有关研究发现，含有全连接层的 CNN 比没有全连接层的 CNN 在性能上表现更好。这可能是因为卷积层由于尺寸小而学习了更多的局部特征，而全连接层则提供了关于如何将这些局部特征连接在一起以产生预期的最终输出的全局图像。图 5-4 显示了用于对图像进行分类

的典型 CNN 架构。

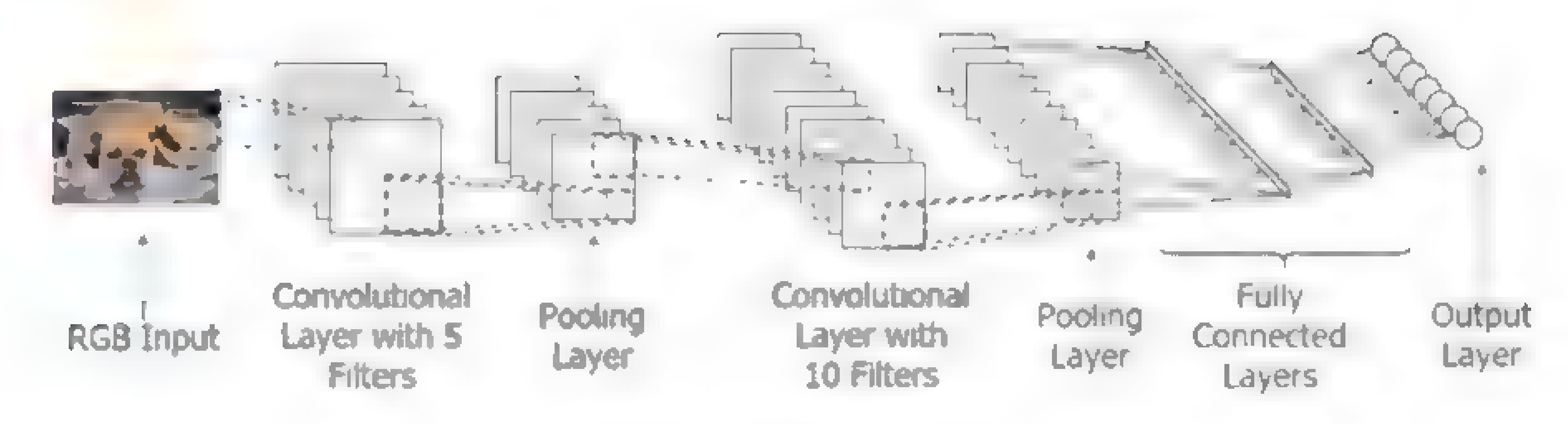


图 5-4 CNN 典型架构

从 5-4 图中可以明显地看出，CNN 在学习期间保留了输入的空间结构。保留空间结构将会使 CNN 利用输入层有价值的空间信息并以较少的参数学习输入。空间信息的价值以图 5-5 为例做简要说明。

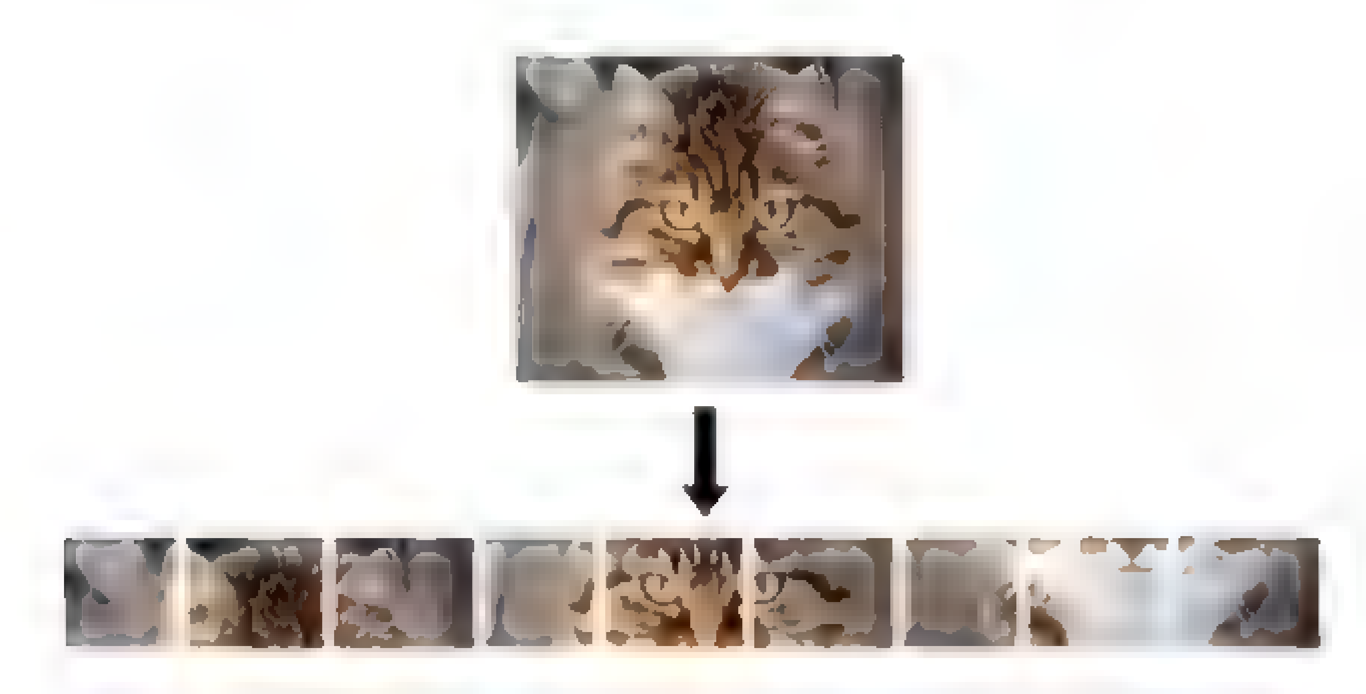


图 5-5 将图像展开成一维向量会丢失一些重要的空间信息

正如我们所看到的那样，当猫的二维图像被释放成为一维向量时，耳朵不再靠近眼睛，鼻子也远离眼睛。这意味着我们在展开过程中破坏了一些有用的空间信息。

我们可以发现，卷积神经网络只不过是各层级网络的功能和形式发生了变化而已，本质上依旧是层级网络。结合图 5-4 可知，卷积神经网络各层级结构如下：

- 输入层/ Input Layer。
- 卷积运算层/Convolutional（CONV）Layer。
- 激励层 / Activation Function Layer。
- 池化层 / Pooling Layer。
- 全连接层 / Fully Connected（FC）Layer。

卷积神经网络是一个多层的神经网络，其基本运算单元包括卷积运算、池化运算、全连接运算和识别运算，如图 5-6 所示。

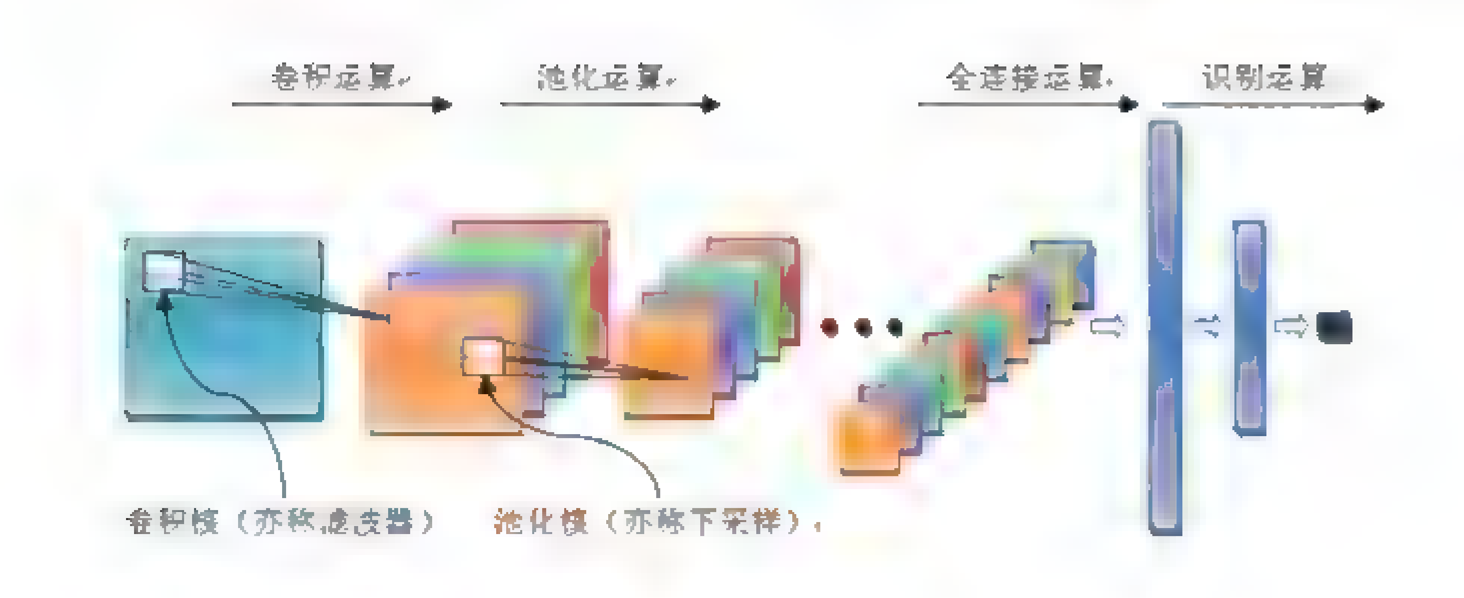


图 5-6　卷积神经网络的基本运算单元结构图

下面针对上面 CNN 的 5 个层级做进一步的解读。

5.2　输入层

数据输入层主要是对原始数据进行预处理工作，具体如下：

（1）去均值处理。我们将对输入样本数据各个维度进行中心化为 0 的处理工作，把输入样本的中心移到坐标系原点，以避免输入数据出现过多偏差，影响训练效果，这就是去均值处理。CNN 只对训练集进行去均值处理，如图 5-7 所示。

（2）归一化处理。把所有数据都归一到同样的取值范围内，即减少因各维度数据取值范围的差异而带来的干扰。

（3）去相关（PCA）/白化处理。去相关就是用 PCA 进行降维工作。白化是对数据样本各个特征轴上的幅度归一化。去相关与白化效果如图 5-8 所示。

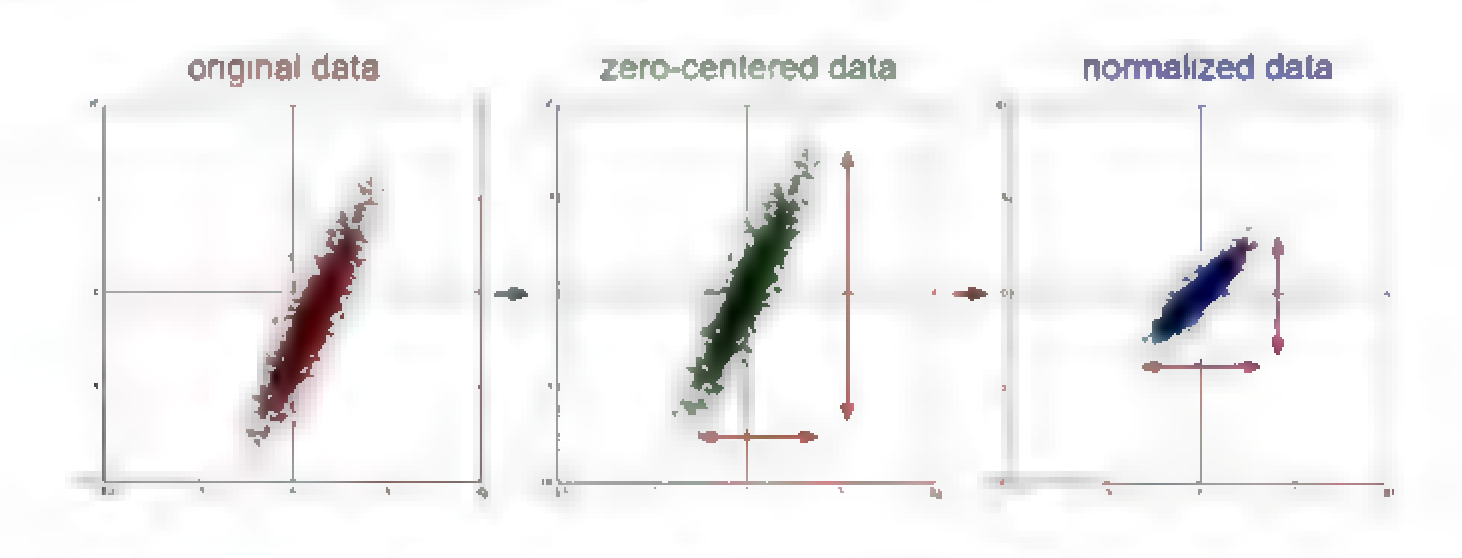


图 5-7　去均值与归一化效果图

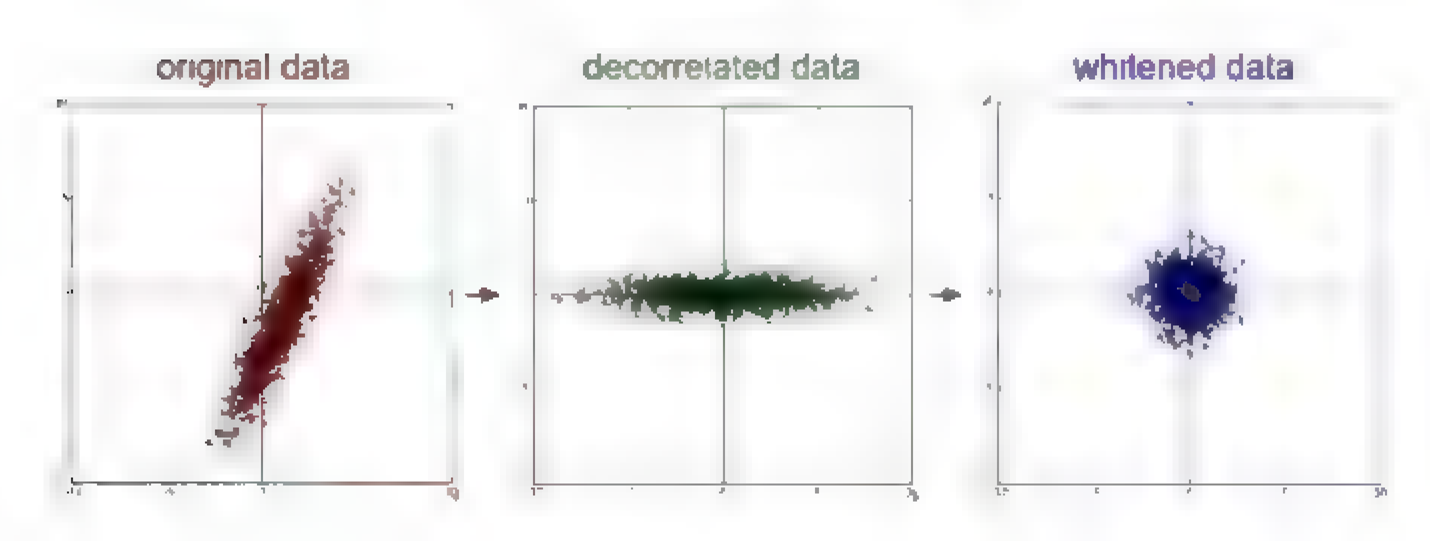


图 5-8　去相关与白化效果图

5.3 卷积运算层

在本节中，我们首先讨论没有步幅（Stride）和填充（Padding）的卷积运算，然后我们将讨论带步幅的卷积运算和带填充的卷积运算，最后会讨论一些被称为转置卷积的内容。

5.3.1 标准卷积

卷积层操作是 CNN 的核心部分，这一层有两个关键的操作：

（1）局部关联。每一个神经元被看作一个过滤器（Filter）。

（2）感受野（Receptive Field）滑动（或称为过滤器滑动），过滤器对局部数据进行计算处理。

对于大小为 $n \times n$ 的输入和 $m \times m$ 的过滤器（又称为滑动窗口或权重值 patch，有的地方也称为神经元），其中 $n>=m$。假设输入是 X，权重值为 W，输出为 H，则在每个位置（i,j）上的输出如式（5.1）所示。

$$h_{(i,j)} = \sum_{k=1}^{m}\sum_{l=1}^{m} w_{(k,l)}\, x_{(i+k-1,j+l-1)} \tag{5.1}$$

这里，$1 \leqslant i,j \ll n-m+1$。$x_{(i,j)}$、$w_{(i,j)}$和$h_{(i,j)}$分别表示 X、W 和 H 的第（i，j）位置处的值。如公式所示，虽然输入大小为 $n \times n$，但在这种情况下，输出大小将为$(n-m+1) \times (n-m+1)$。此外，m 被称为过滤器大小。下面让我们通过一个可视化的例子进行解读，如图 5-9 所示。

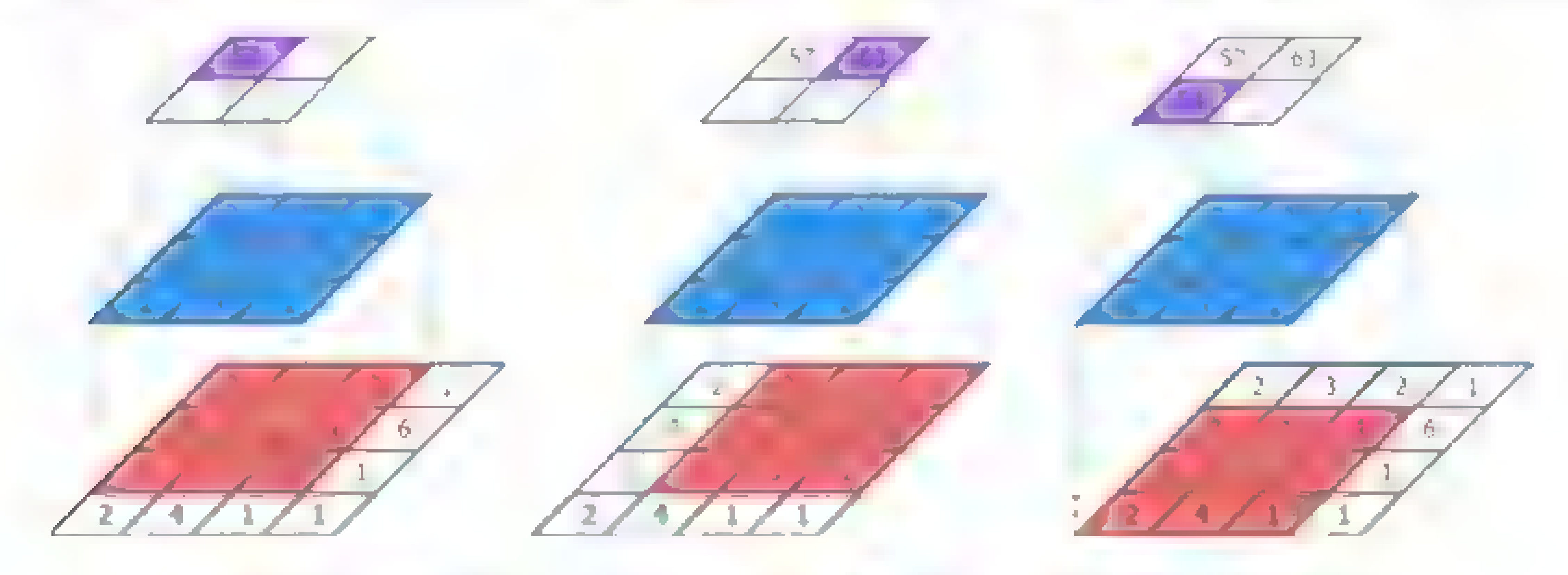


图 5-9 过滤器（m）= 3 和单位步幅且没有填充的卷积运算示例

这里以图 5-9 中最左侧图示为例，过滤器矩阵大小为 $m \times m$，即 3× 3 的矩阵（中间位置的蓝色矩阵），对于输入大小$(n \times n)$为 4×4 的矩阵（最底部矩阵）而言，将会得到的输出矩阵大小为$(n-m+1) \times (n-m+1)$，即$(4-3+1) \times (4-3+1)=2 \times 2$ 的输出矩阵（最上面的矩阵）。利用上面的公式，我们给出经过卷积运算后输出的值，具体算法如下：

```
2*2+3*3+2*1+3*1+1*2+4*3+1*2+5*4+3*1=57
```

这里的步长为 1，且输入矩阵没有被填充，以此类推，让过滤器在输入层上逐步滑动，最终对整个输入层进行卷积运算得到一个确认的 2×2 输出矩阵。图中的顶部矩形（卷积运算产生的输出）有时被称为特征映射。

当然，对于图像水平和垂直边缘检测而言，这里可以通过水平过滤器和垂直过滤器来实现。目前，对于一些常用的过滤器也存在着不同的争论，但是在 CNN 中我们把这些过滤器当成要学习的参数，CNN 训练的目标就是去解析过滤器的参数。

5.3.2　带步幅的卷积

在 5.3.1 节的示例中，我们只是将过滤器移动了 1 个步幅，其实我们也可以做更多步幅（步长）的移动。在公式（5.1）的基础上做修改，包括 s_i 和 s_j 的步幅，我们可以得到带步幅的卷积情况下的输出大小，如公式（5.2）所示。

$$h_{(i,j)}=\sum_{k=1}^{m}\sum_{l=1}^{m}w_{(k,l)}\,x_{((i-1)\times s_i+k,(j-1)\times s_j+l)} \qquad (5.2)$$

这里，$1\leqslant \quad i\ll floor[(n-m)/s_i]+1\wedge floor[(n-m)/s_j]+1$。

在这种情况下，随着s_i和s_j的大小增加，其对应输出将变小。下面比较图 5-9（stride = 1）和图 5-10（stride = 2）中给出的不同步幅的影响。

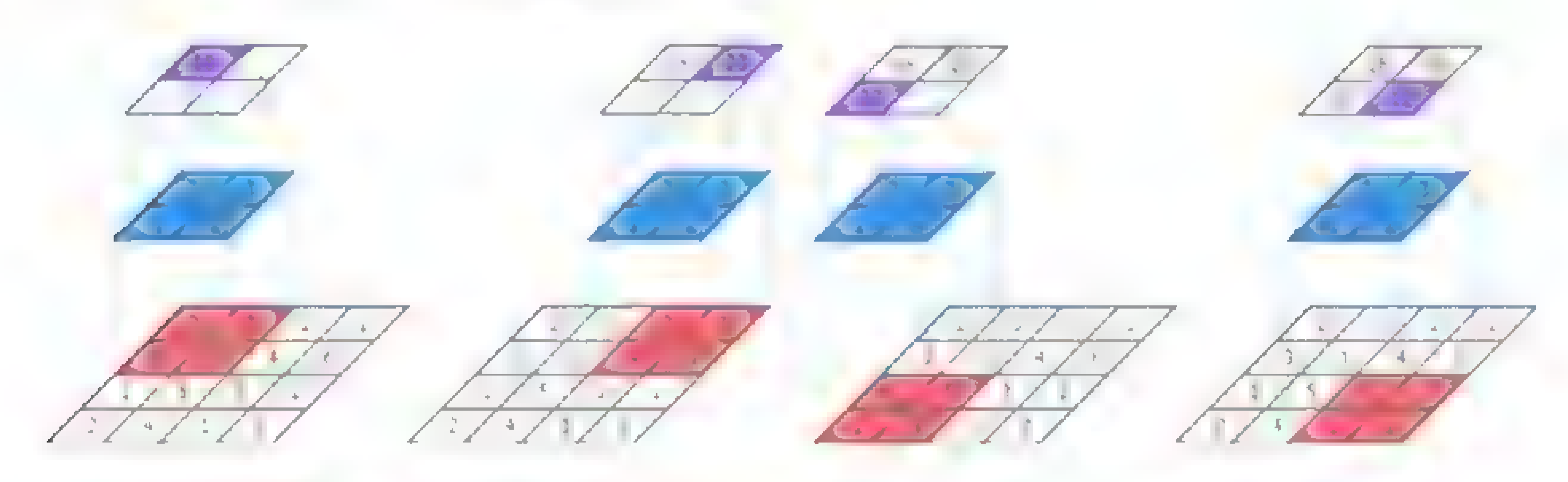


图 5-10　过滤器（m）= 2 和 stride = 2 且没有填充的卷积运算示例

具体计算过程与 5.3.1 节中类似，这里就不再重复给出了。

显然，由图 5-10 可知，使用步幅（Stride）进行卷积运算有助于降低输入样本数据的维度，与池化运算功能类似，所以有时也会使用带有步幅的卷积代替 CNN 中的池化运算，因为它降低了计算的复杂度。

5.3.3　带填充的卷积

在 5.3.1 和 5.3.2 节中，我们分别使用一个 3×3 的过滤器和一个 2×2 的过滤器对 4×4 的输入样本数据进行卷积运算，均得到了一个 2×2 的输出，输出数据大小为$(n-m+1)\times(n-m+1)$。但是，这样卷积运算的缺点也很明显，假设输入的样本是图像数据，那么卷积图像的大小会不断缩小，且图像边界的元素只被对应的一个输出所使用，因此，图像边缘处的像素在输出时会减少，显然这样的运算使得原图像边缘的信息丢失很多。为了解决这个问题，我们引入了填充（Padding）操作，即在图像卷积运算之前沿着图像边缘均用 0 值进行填充。假设步幅（Stride）为 1，输出大小的如公式（5.3）所示。

$$h_{(i,j)} = \sum_{k=1}^{m} \sum_{l=1}^{m} w_{(k,l)} x_{(i+k-(m-1),j+l-(m-1))} \tag{5.3}$$

这里：$1 \leqslant i,j \ll n$；若 $i,j>n$ 或者 $i,j<0$，则 $x_{(i,j)} = 0$。

图 5-11 给出了滤波（m）=3、步幅（s）= 1 且填充值为 0 的卷积运算。

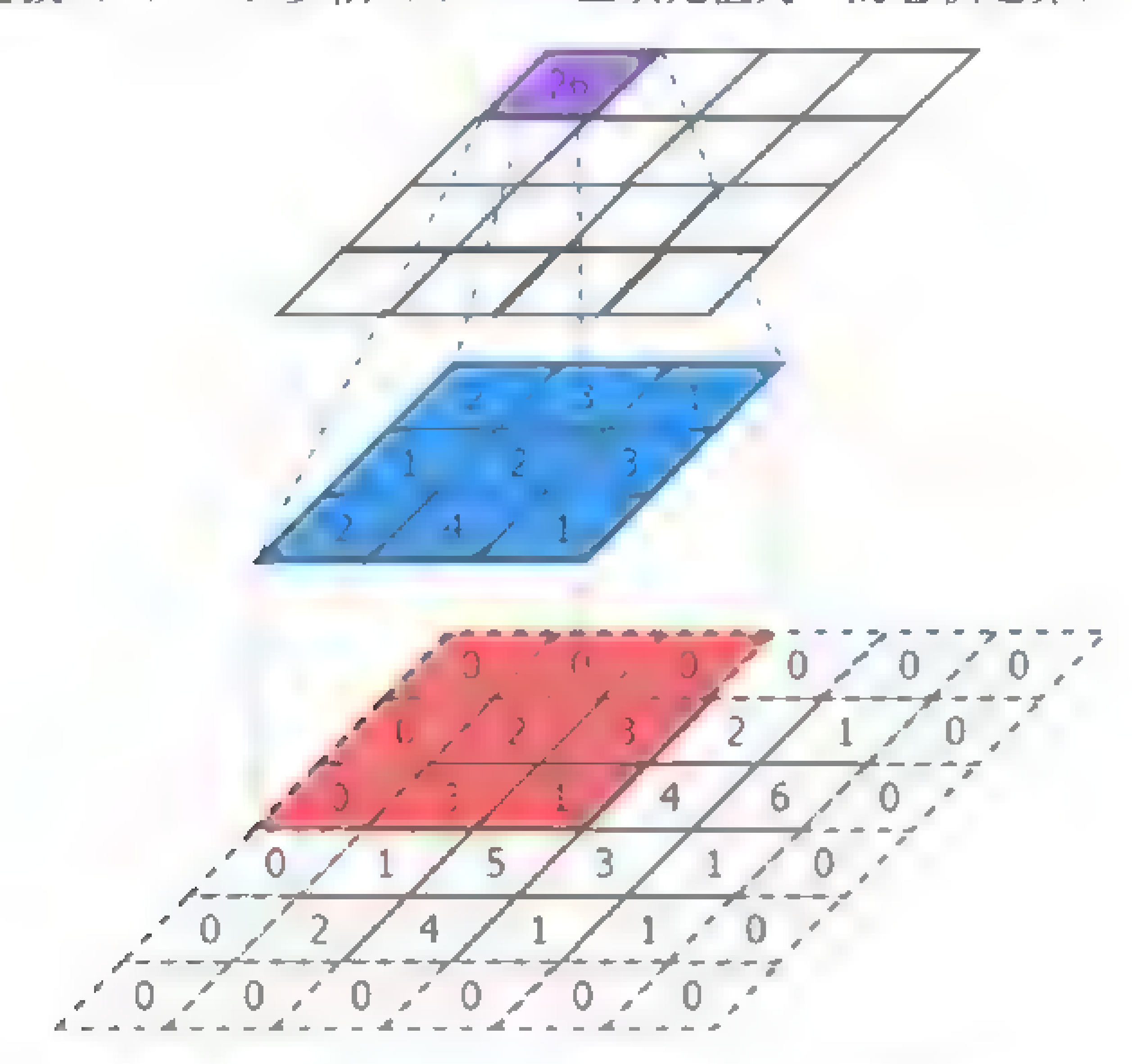


图 5-11　具有过滤器大小（$m=3$）、步幅（$s=1$）和零填充的卷积运算

这里过滤器矩阵大小为 3×3，而填充之后，输入矩阵大小为 6×6，则输出矩阵大小变为(6-3+1）×（6-3+1）=4×4。以图 5-11 中输入层左上侧为例，计算卷积之后对应的输出值大小：

0×2+0×3+0×1+1×0+2×2+3×3+0×2+3×4+1×1=26

以此类推，让过滤器在输入层上逐步滑动，最终对整个输入层进行卷积运算得到一个确认的 4×4 输出矩阵。

5.3.4　转置卷积

尽管卷积运算在数学方面看起来很复杂，但它可以简化为矩阵乘法，这就出现了转置卷积（Transposed Convolution）。简单理解就是，对于卷积在神经网络结构中的正向和反向传播做相反的运算。现在有如图 5-12 所示的卷积运算。

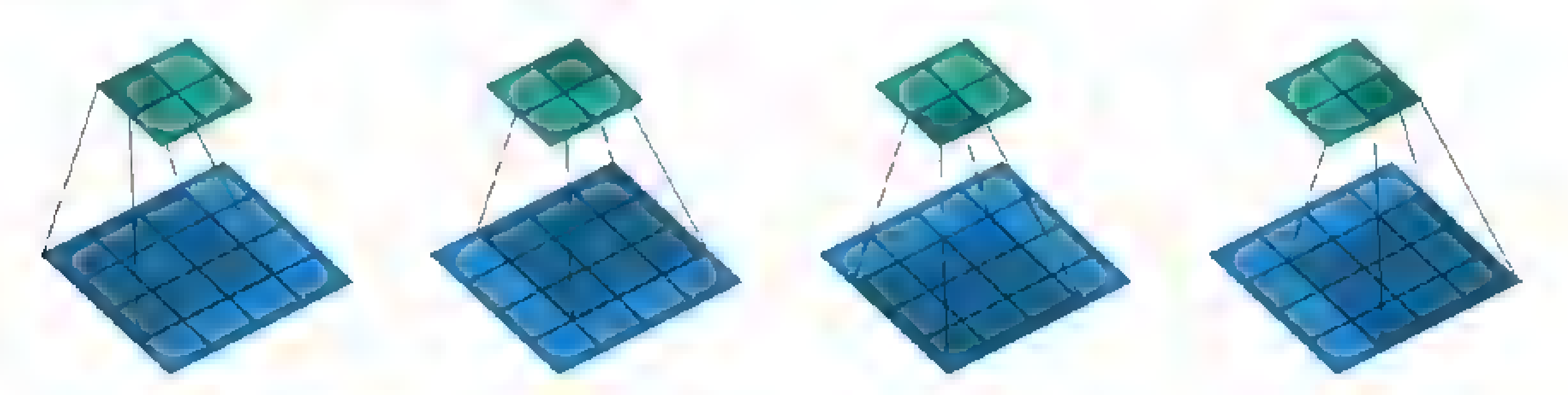


图 5-12　单位步幅在 4×4 输入上 3×3 滤波且无填充（$n=4$，$m=3$，$s=1$ 且 $p=0$）

以图 5-11 中表示的卷积为例，通过线性运算将输入和输出从左到右、从上到下展开，则输入 $\boldsymbol{X}$ 矩阵被平坦化为 16 维向量，输出 $\boldsymbol{H}$ 被平坦化为 4 维向量，该向量随后被重新整形为 2×2 输出矩阵，也可以得到展开后卷积的稀疏矩阵 $\boldsymbol{A}$，如下所示。

$$x^{(16,1)} = x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4,\dots}, x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}$$

$$A^{(4,16)} = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{1,1} & w_{1,2} & w_{1,3} & 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & w_{3,1} & w_{3,2} & w_{3,3} \end{pmatrix}$$

$$H^{(4,1)} = A^{(4,16)} X^{(16,1)}$$

其中，非零元素是权重值$w_{(i,j)}$（i 和 j 分别是过滤器的行和列）。

注意，由于本章节的所有索引都是从 1 开始，而不是从 0 开始，因此上方矩阵是从左上角$w_{1,1}$开始到右下角$w_{3,3}$结束。读者也可以查阅一下 Vincent Dumoulin 等人写的《*Guide to Convolution Arithmetic for Deep Learning*》，文章里面关于转置卷积的论述很好，涉及的索引默认是从 0 开始的。

现在，通过将输出$\boldsymbol{H}^{(4,1)}$重新整形为$\boldsymbol{H}^{(2,2)}$，我们获得了卷积输出。现在让我们将这个结果转回到 n 和 m：

通过将输入$\boldsymbol{X}^{(n,n)}$平坦化（展开）为$X^{(n^2,1)}$，并通过从 w 创建矩阵$\boldsymbol{A}^{((n-m+1)^2,n^2)}$，如前面所示，我们获得$\boldsymbol{H}^{((n-m+1)^2,1)}$，然后将其重新整形为$\boldsymbol{H}^{(n-m+1,n-m+1)}$。接下来，为了获得转置卷积，我们只需转换 $\boldsymbol{A}$ 就可以得到如下结果：

$$(\hat{X})^{(n^2,1)} = (A^T)^{(n^2,(n-m+1)^2)}(H)^{((n-m+1)^2,1)} \hat{X}^{(n^2,1)} = (A^T)^{(n^2,(n-m+1)^2)} H^{((n-m+1)^2,1)}$$

这里，$\hat{X}$是转置卷积的结果输出。

关于转置卷积就介绍到这里，下面我们将要介绍卷积神经网络的另外一个重要特性——参数共享机制。

5.3.5　参数共享机制

CNN 最大的特点是卷积的权重值共享结构，可以大幅度减少神经网络的参数个数，防止过拟合且同时又降低了神经网络模型的复杂程度，所以我们这里探讨一下 CNN 的权重值共享机制问题。所谓权重值共享，就是针对一张给定输入图像用一个过滤器（Filter）去扫描，过滤器里面的数叫作权重值，如果这张图像的每个位置是被同样的过滤器扫描，则对应权重值是一样的，也就可以说是共享的。举例说明，如图 5-13 所示。

由图 5-13 可知，左侧有一张 1000×1000 像素的图像且有 100 万个隐藏层神经元，那么如果是全连接的话（每个隐藏层神经元都连接到图像中的每一个像素点），这里就有 1000×1000×1000000=10^{12}个连接，那么权重值参数个数也就是10^{12}个，显然，这时的参数个数太多，下一步就要减少参数。

图像的空间联系是局部的，与一个人通过一个局部的感受野去感受外界图像类似，那么每一个

神经元就不必对全局图像做感受而只对局部图像区域做感受即可。这样一来，输出层的每一个像素就只和输入层图片的一个局部相连，显然需要参数的个数就会大大减少。当然，这些感受到的局部图像区域会在更高层被综合起来，从而得到全局的图像信息。如图 5-13 右图所示，局部感受野是 $f \times f$ 即 10×10，因为隐藏层只需要与这个 $f \times f$(10×10)的局部图像相连接，所以 100 万个隐藏层神经元就只有 1 亿个连接，即10^8个参数，显然参数个数比原来少了不少。其实，这样的参数个数还是很多，我们下面继续降低参数个数。

其实，图像还有另外一个重要特性，那就是图像底层特征与特征在图像中的位置无关。例如，无论是图像中间的边缘特征还是图像边角处的边缘特征，我们都可以用类似的特征提取器进行特征提取。对于主要针对底层特征的前几层网络，可以把上面局部全连接层中的每一个 $f \times f$ 的小方块所对应的权重值进行共享，以此进一步减少参数个数。换句话讲，输出层的每一个像素，是由输入层对应位置 $f \times f$ 的局部图像与同一组 $f \times f$ 的权重值做内积，再由非线性单元计算得来的。最终，无论图像起初大小如何，只需要 $f \times f$ 个参数就可以了。因为一组 $f \times f$ 参数只能得到一张特征图（Feature Map），所以有几张特征图再乘以几就可以得到最终的参数个数。这里，以图 5-13 为例，因为隐藏层每一个神经元都连接 $f \times f$(10×10)的局部图像区域，也就是说每一个神经元存在 10×10=100 个连接权重值参数，每个神经元用的是同一个卷积核去卷积图像，所以这里只需要 100 个权重值参数就可以了，这就是权重值共享。当然，假设有 100 组权重值参数，那么我们总共需要 100×100＝10000 个参数就可以了。当然，像人脸图像这样的高级特征一般是与位置有关的。眼睛和嘴的位置不同，在高层进行处理时，不同位置需要用不同的神经网络权重值。由于这不是卷积运算的内容，因此这里就不做过多阐述了。

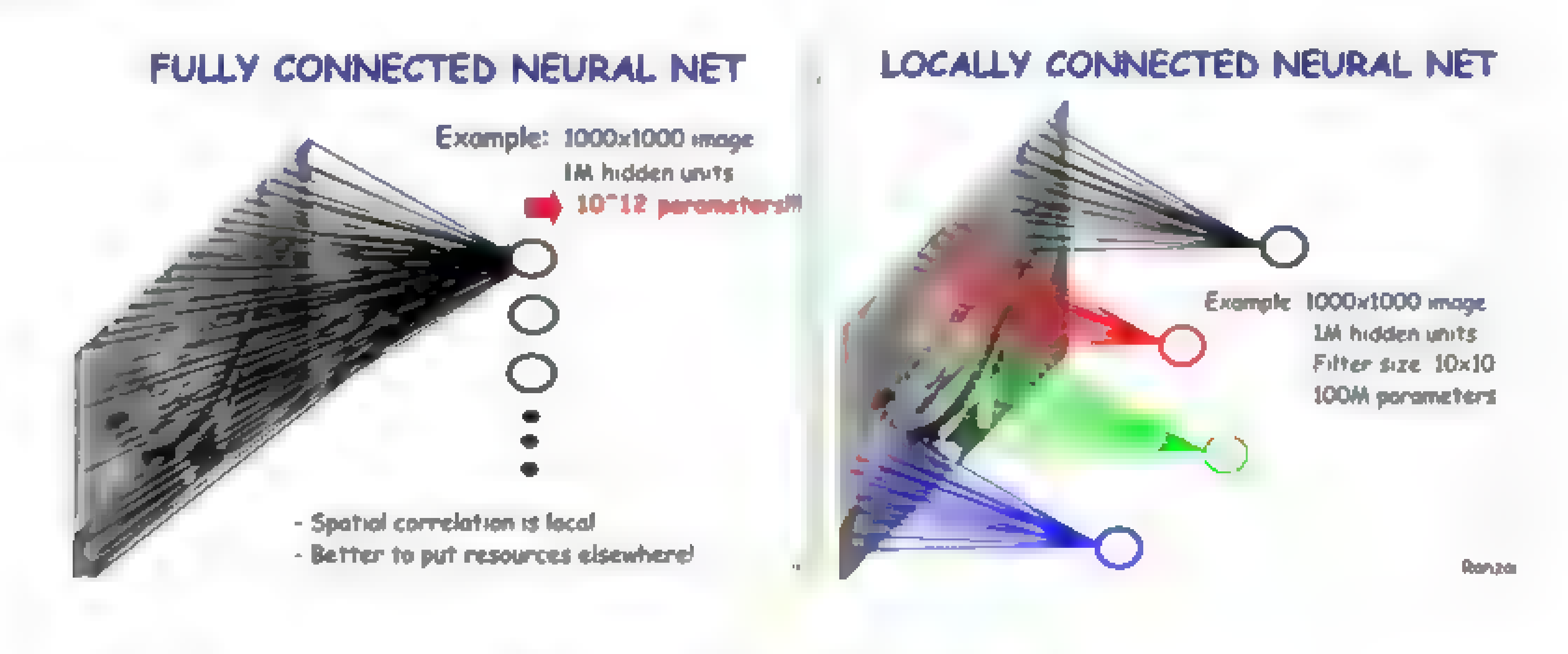


图 5-13 参数共享机制示例图

对于 RGB 的彩色图像而言，将原来图像的宽度×高度调整为宽度×高度×3（3 是通道数）即可。其实，更确切一点，就权重值共享而言，它只是在每一个过滤器（Filter）上的每一个通道（Channel）是共享的。

至此，我们完成了卷积运算的讨论，下一步我们将开启池化运算。

5.4　激活函数

5.4.1　常见激活函数及选择

常用的激活函数如图 5-14 所示。

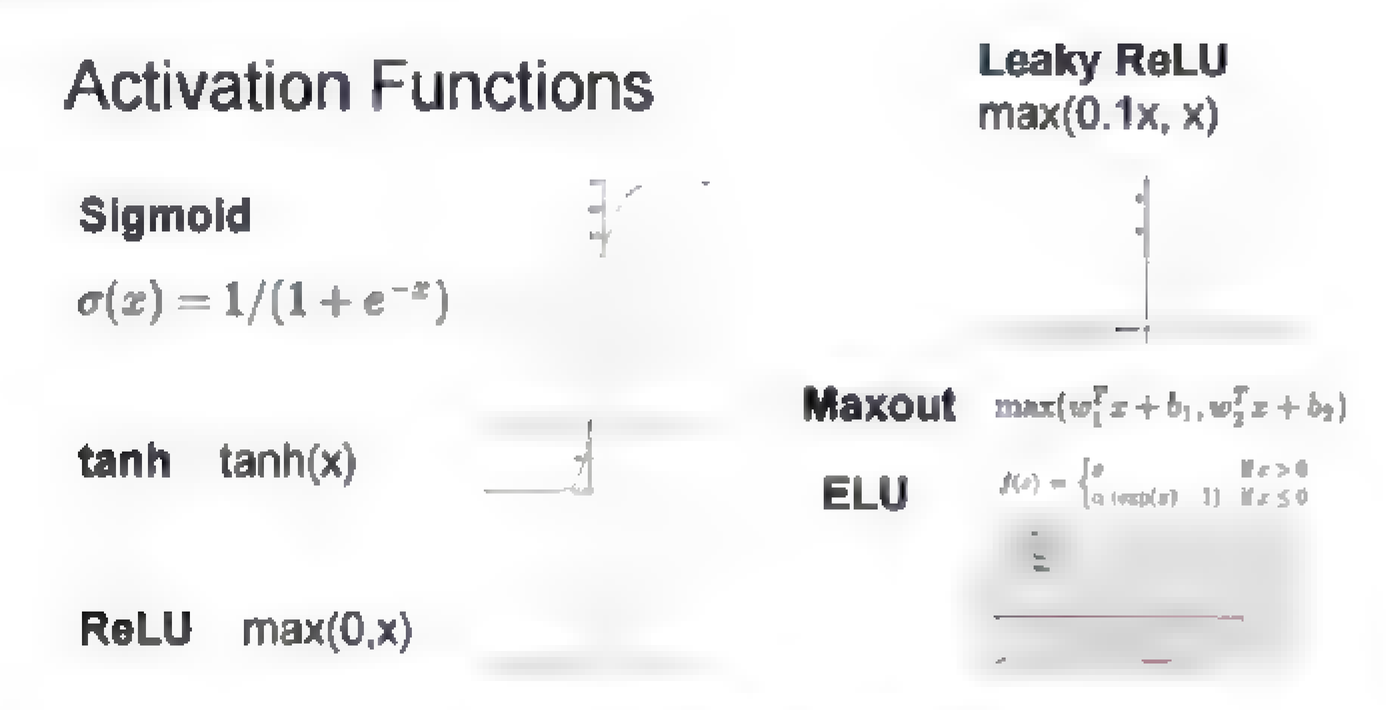


图 5-14　常见的激活函数汇总

在对卷积层输出的结果进行非线性映射处理时，我们多数会摒弃 Sigmoid、tanh 函数，所以这里选择 ReLU、Leaky ReLU、Maxout、ELU 四种非线性激活函数。

有了非线性激活函数，我们下面给出神经元处理的相关流程（结合图 5-15 进行简要说明，当然这里都是非线性函数）。

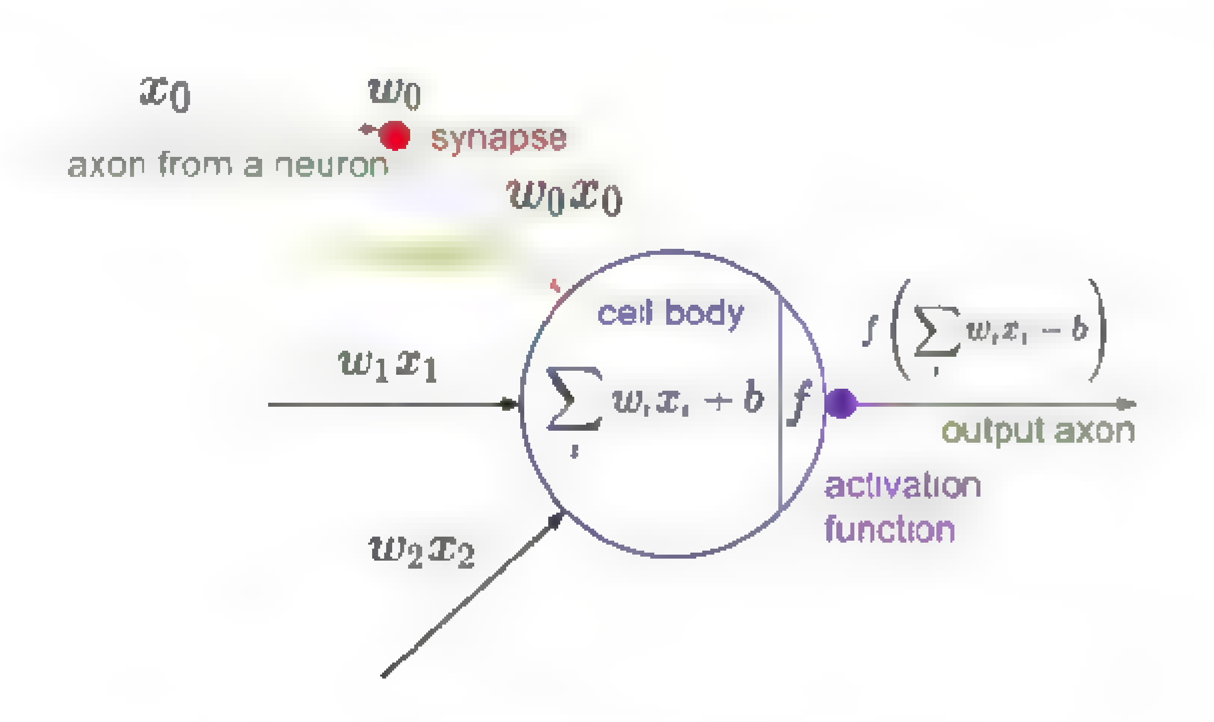


图 5-15　数学模型中神经元处理的一般流程

数学模型中神经元处理的一般流程如下：

（1）输入层 X 与对应权重值做内积（Dot Product，也称为点积）处理。

（2）将激活函数作用于上面的内积结果。

（3）激活函数输出相应的结果。

5.4.2 各个非线性激活函数对比分析

1. ReLU 函数

最近几年，ReLU（Rectified Linear Unit）激活函数在深度学习领域确实比较受欢迎，它的译名是修正线性单元，这个函数其实是一个非线性操作，主要起源于传统激活函数、脑神经元激活频率研究、稀疏激活性。

ReLU 函数的数学表达式和对应图像如图 5-16 所示。

$$f(x) = max(0, x)$$

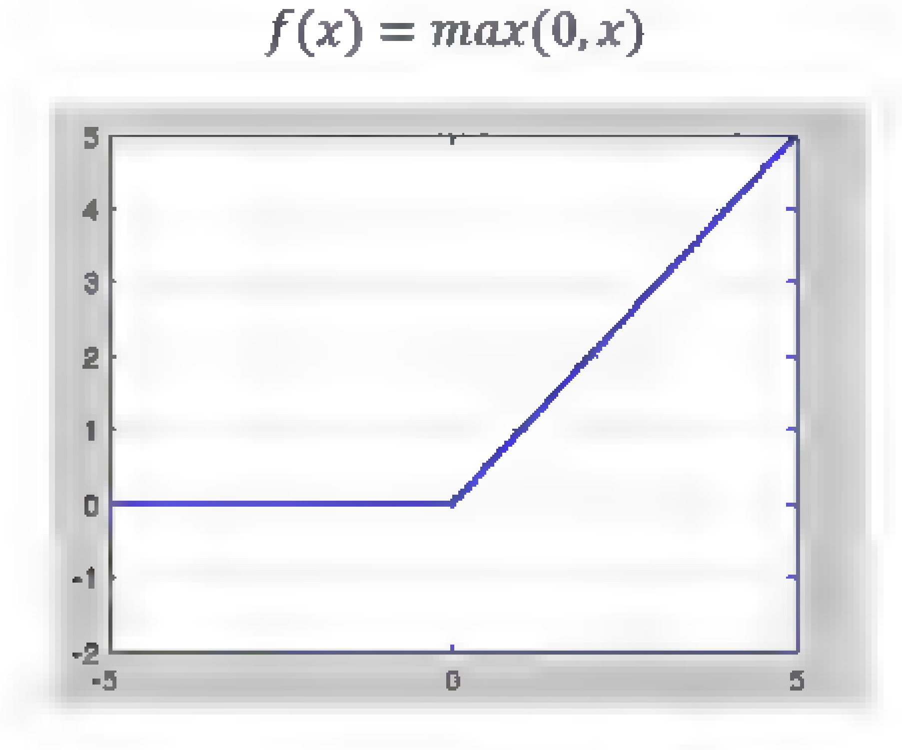


图 5-16 ReLU 函数示意图

我们发现，当 $x<0$ 时，ReLU 是硬饱和；当 $x>0$ 时，不存在饱和区问题。因此，激活函数 ReLU 可以在 $x>0$ 时保持梯度不衰减，进而缓解梯度消失问题。这就允许我们直接以监督的方式训练深度神经网络，而无须依赖无监督的逐层预训练。随着训练的不断深入，存在部分输入会落入硬饱和区，导致对应权值无法更新。这种现象被称为“神经元死亡”。与 sigmoid 类似，ReLU 的输出均值也大于 0，偏移现象和神经元死亡会共同影响网络的收敛性。

综上所述，我们对 ReLU 函数的优缺点做简单的总结，具体如下：

（1）优点

- 没有饱和区，收敛速度比较快。
- 梯度计算起来比较简单，只需要一个阈值便可以获得激活值。

（2）缺点

- ReLU 有时表现得很脆弱，甚至进入死亡区。
- ReLU 激活函数作用后的输出结果均值不是 0。

关于梯度爆炸和梯度消失的理解，简单来说，就是在反向传播算法过程中，因为使用了矩阵求导链式法则，这里存在一长串连乘，当连乘数字在每一层均小于 1 时，就会引起梯度越靠前乘越小，进而梯度消失；反之，梯度越靠前乘越大，进而梯度发生爆炸。所以，有时 ReLU 的表现很脆弱。如果在变量更新很快的情况下还没有找到最佳值，那么进入小于 0 的分段就会迫使梯度消失（梯度值变为 0），也就是这时部分输入会落入硬饱和区，以至于无法实现继续再更新，所以我们在使用

ReLU 激活函数时应该控制好学习率（Learning Rate），否则神经元会有很大的概率进入死区。这里再强调一次，ReLU 激活函数有时表现得很脆弱，但从总体来看 ReLU 激活函数至少解决了部分梯度消失问题，本身还是有很好的应用价值。

2. ReLU 的变体函数

针对在 $x<0$ 的硬饱和问题，我们对 ReLU 进行相应的改进，便得到了改进后的激活函数 ReLU 变体——PReLU（ParametricRectifier，参数修正器），其数学表达式如下，图像如图 5-17 所示。

$$f(x_i) = max(\alpha_i, x_i)$$

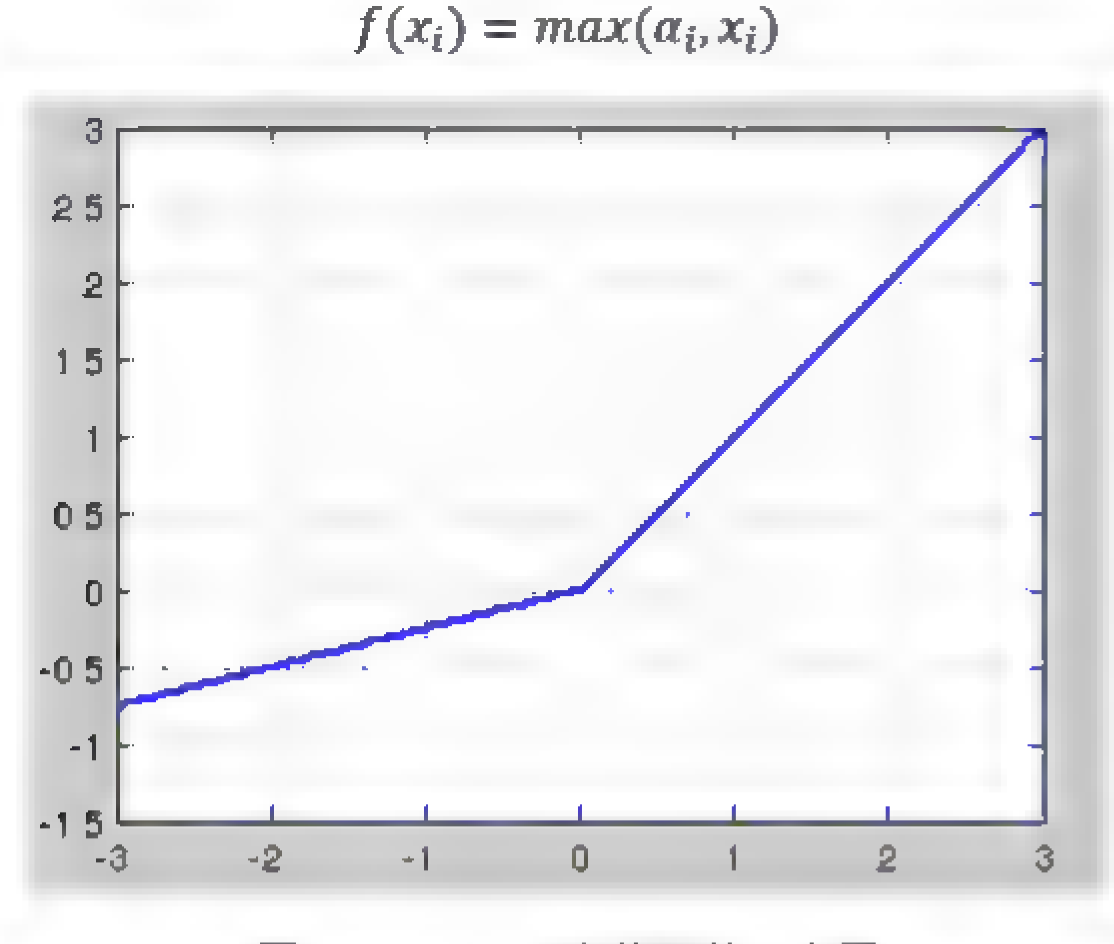


图 5-17　ReLU 变体函数示意图

这里，α_i是可以变化且能进行学习的，i 表示通道数。

当α_i的值退化到 0 时，PReLU 函数就退化为 ReLU 函数。

当α_i取一个非常小的固定值时（比如为 0.01），PReLU 函数退化为 Leaky ReLU（LReLU）函数。在这种情况下，我们能够做到既修正了数据分布，又保留了一些负轴上的值，进而保证了负轴信息不会全部丢失。

当α_i从高斯分布中随机产生时，PReLU 函数便成了 Randomized Leaky ReLU 函数。它的基本原理就是，在训练过程中，α_i在一个高斯分布中随机产生，然后对测试过程进行修正。

另外，当不同的通道使用不同的α_i时，参数是很少的。而 BP 更新α_i时，采用了带动量的更新方式。

3. Maxout 函数

2013 年，Goodfellow 将 maxout 和 dropout 结合后在 ICML2013 上提出了 Maxout 函数，并宣称在 MNIST、CIFAR-10、CIFAR-100、SVHN 这 4 个数据上都取得了当时最好（start-of-art）的识别率。Maxout 函数的数学表达式如下：

$$f_i(x) = max_j \in_{[1,k]} z_{i'j}$$

或者

$$f_i(x) = max(w_1^T x, +b_{1,} w_2^T x, +b_{2_}, w_n^T x, +b_n,)$$

仔细查看 Maxout 函数，不难发现 Maxout 的网络可以接近任意连续函数，几乎可以涵盖所有激活函数。当 $w_2, b_2, \ldots, w_n, b_n$ 取 0 值时，便退化为 ReLU 函数。它的拟合能力非常强，可以拟合任意凸函数。

最后，我们会发现，Maxout 既有 ReLU 函数的优点（计算简单，不会饱和，避免梯度消失）又规避了 ReLU 函数的不足（神经元死亡）。同时，Maxout 函数也不是没有缺点，缺点就是增加了参数和计算量。

4. ELU 函数

指数线性单元（Exponential Linear Unit，ELU 函数）是由 Djork-Arné Clevert、Thomas Unterthiner、Sepp Hochreiter 在《*Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*》（最新修订日期为 2016 年 2 月 22 日）一文中提出的，其数学表达式如下，图示如图 5-18 所示。

$$f(x) = \begin{cases} \alpha(e^x - 1), & x \leq 0 \\ x, & x > 0 \end{cases}$$

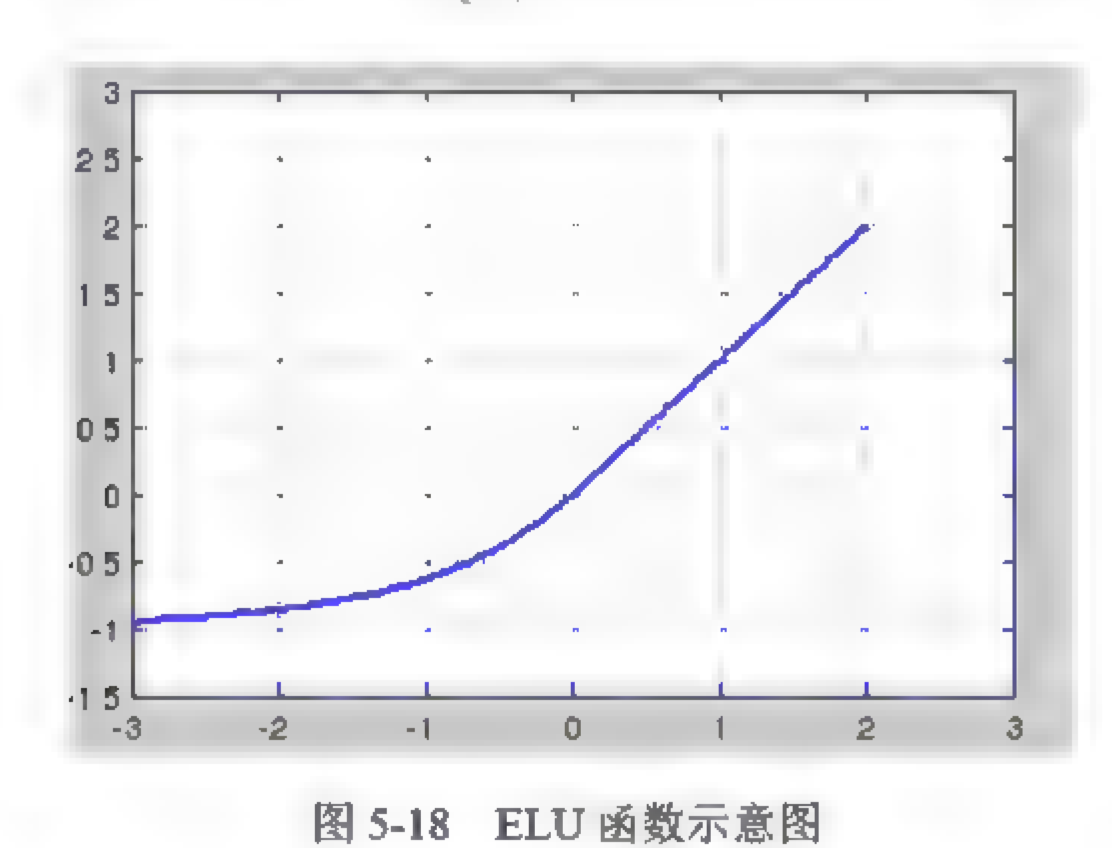


图 5-18　ELU 函数示意图

ELU 函数其实是整合了 sigmoid 和 ReLU 两个函数，也可以理解为对这两个函数的一般归纳，于是便使得其具有左侧软饱和性、右侧无饱和性的特性。也就是说，ELU 函数的右侧线性部分能够缓解梯度消失而左侧的软饱和又对输入的内容变化或噪声更有鲁棒性，且 ELU 的输出均值近似为 0，因此其收敛性更强。ELU 函数中含有指数，所以其计算量还是比较大的。

至此，我们已经将激活函数介绍完毕，下面我们将对池化层进行解读。

5.5　池化层

5.5.1　理解池化

由于图像具有“静态型”属性，因此我们会继续使用经过卷积层、激活后的特征，同时也说明一个图像区域的特征有很大概率同样适用于另一个区域。所以，我们在描述一个图像时，就可以对图像的不同区域特征进行聚合统计（一般是矩形区域上的统计）。这种聚合统计操作就是池化。池

化输出的是邻近区域的概括统计量。池化有最大池化、平均池化、滑动平均池化、L^2范数池化等，下面主要介绍常见的最大池化和平均池化运算或操作。

5.5.2　池化作用

（1）不仅可以在保留主要特征的同时减少参数个数（降低维度，类似 PCA）和计算量，还可以防止过拟合现象的出现。

前面我们卷积运算得到了相关特征（Feature），接下来就要用这些特征去进行分类。当然可以使用所提取到的特征去训练我们的分类器，但是这样会使得计算成本很高。就拿 softmax 分类器来讲，现在有一个 100×100 像素的图像，假如我们已经学习到了 500 个定义在 8×8 输入上的特征，则每一个特征和图像卷积都会获得一个(100-8+1) ×(100-8+1)=8649 维的卷积特征，又因有 500 个特征，所以每个样例(example)就会得到一个(100-8+1) ×(100-8+1) ×500=4324500 维的卷积特征向量。显然，去学习一个拥有超过 400 万维特征向量的分类器难度不小，且在这种情况下容易出现过拟合现象。

（2）池化运算可以使得层次网络对输入的图像中更小的变化、冗余、变换维持不变性。输入的微小冗余将不会改变池化的输出，因为在局部区域中我们使用了最大化、平均值等运算。

（3）可以帮助我们获得图像最大程度上的特征不变性（Invariance）。这种不变性包括平移（Translation）、旋转（Rotation）、尺度（Scale）方面的不变性。（注意，卷积是对输入平移不变性，池化是对特征平移不变性。）

因此，池化运算比没有填充的卷积引起的自然维数降低更受欢迎，因为我们可以决定在哪里减小池化层的输出大小，而不是每次强制它发生。实际上，没有填充的强制降维在很大程度上会限制我们在 CNN 模型中使用的层数。

5.5.3　最大池化

最大池化（Max Pooling）运算是在输出层上选择已定义内核中的最大元素作为结果来输出。最大池化移位就是输入层上的窗口（过滤器在输入层上移动的投影方块），在移位上选择最大的元素作为一个输出。下面给出池化方程的定义，并以图 5-19 所示为例。

池化方程式：

$$h_{i,j} = max(x_{((i-1)\times s_i+1,(j-1)\times s_j+1)}, x_{((i-1)\times s_i+1,(j-1)\times s_j+2)}, \dots, x_{((i-1)\times s_i+1,(j-1)\times s_j+m)},$$
$$x_{((i-1)\times s_i+2,(j-1)\times s_j+1)}, \dots, x_{((i-1)\times s_i+2,(j-1)\times s_j+m)}, \dots, x_{((i-1)\times s_i+m,(j-1)\times s_j+1)}, \dots,$$
$$x_{((i-1)\times s_i+m,(j-1)\times s_j+m)})$$

这里，公式是包含带步幅的池化运算，且

$$1 \leqslant i \leqslant floor[(n-m)/s_i]+1 \quad 1 \leqslant j \leqslant floor[(n-m)/s_j]+1$$

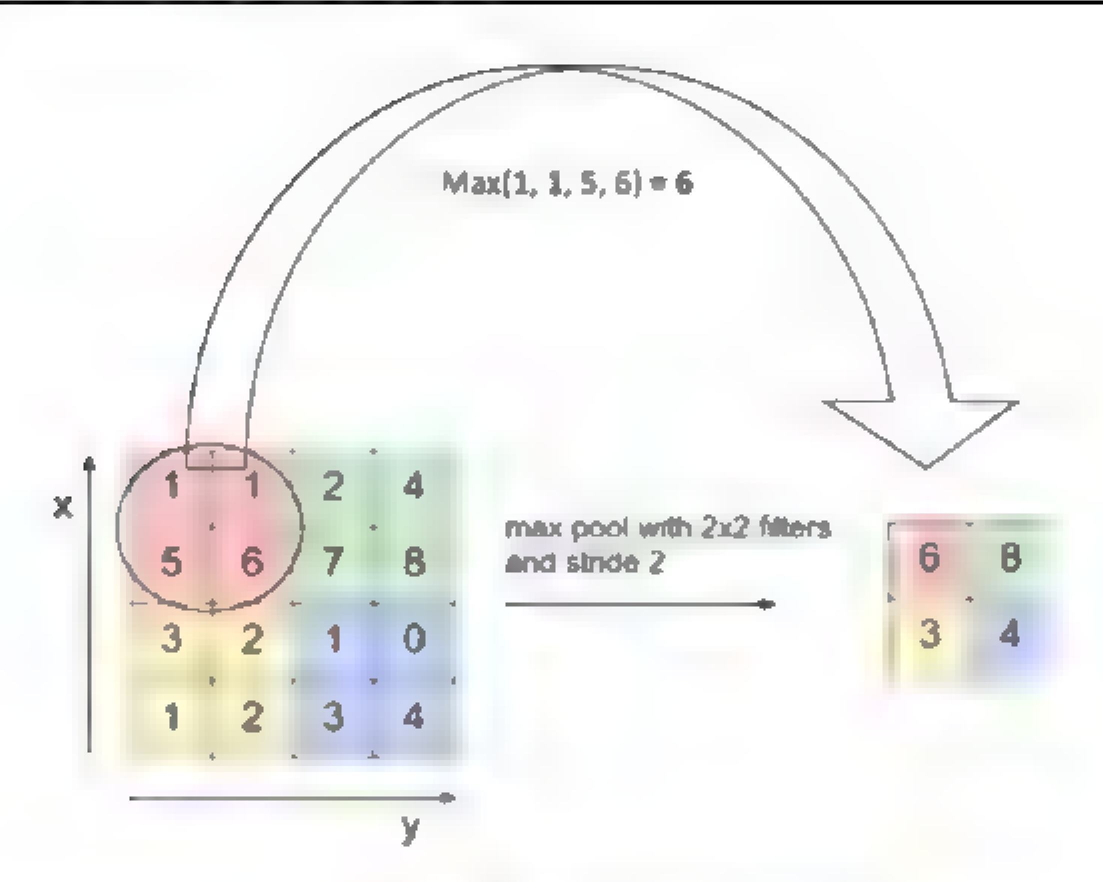


图 5-19　最大池化运算示意图

在图 5-19 中，步长为 2、过滤器为 2×2，在输入层 4×4 矩阵上滑动后，将输入层分成 4 部分（已标注不同颜色），在每个区域中选择值最大的元素作为输出，就会得到右侧的输出结果的汇总。

5.5.4　平均池化

平均池化（Average Pooling）的工作方式类似于最大池化，只是不采用最大值，而是采用内核中所有输入的平均值。考虑以下公式：

$$h_{i,j}=\frac{x_{i,j},x_{i,(j+1)},\cdots,x_{i,(j+m-1)},x_{(i+1),j},\cdots,x_{(i+1),(j+m-1)},\cdots,x_{(i+m-1),j},\cdots x_{(i+m-1),(j+m-1)}}{m\times m}$$

$\forall i \quad 1, j \leqslant n-m+1$

平均池运算如图 5-20 所示。

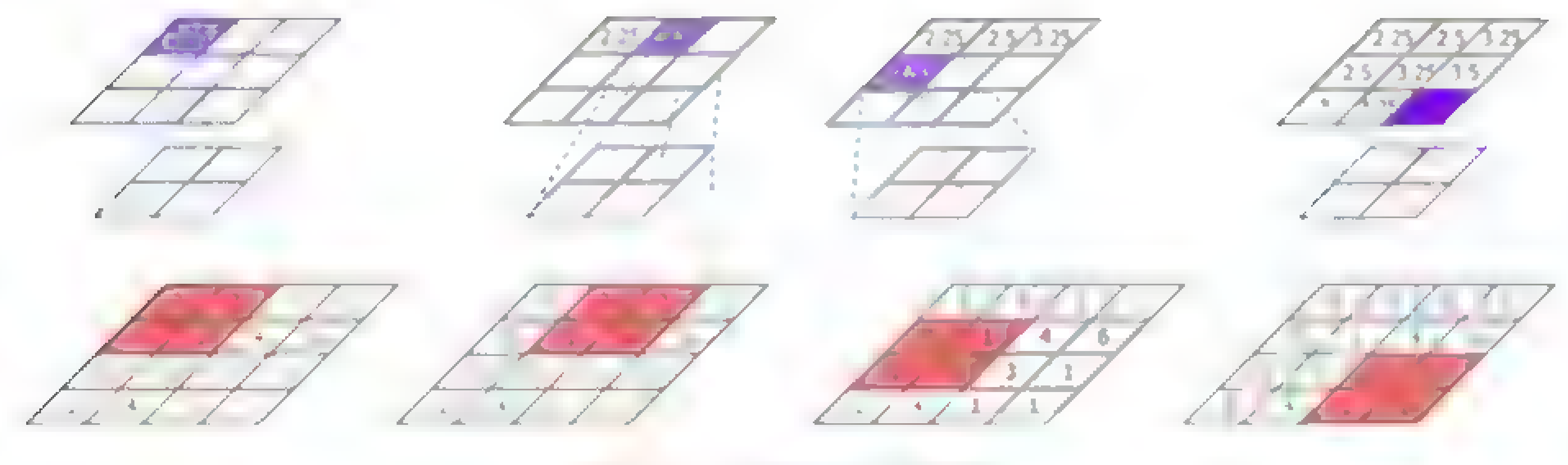


图 5-20　平均池化示例图

5.6　全连接层

卷积层和池化层的输出给出了输入图像的高级特征，而全连接层的目的是为了对这些特征进行维度上的改变并得到每个分类类别对应的概率值。稍微正式一点讲，全连接层是从输入到输出的完全连接的权重值集合，这些全连接的权重值能够在从每个输入连接到每个输出时学习全局信息。具

有完全连接性的这些层允许我们将全连接层之前卷积层所学习的特征从全局角度组合起来，从而产生有意义的输出。下面给出全连接层的大致直观图，如图 5-21 所示。

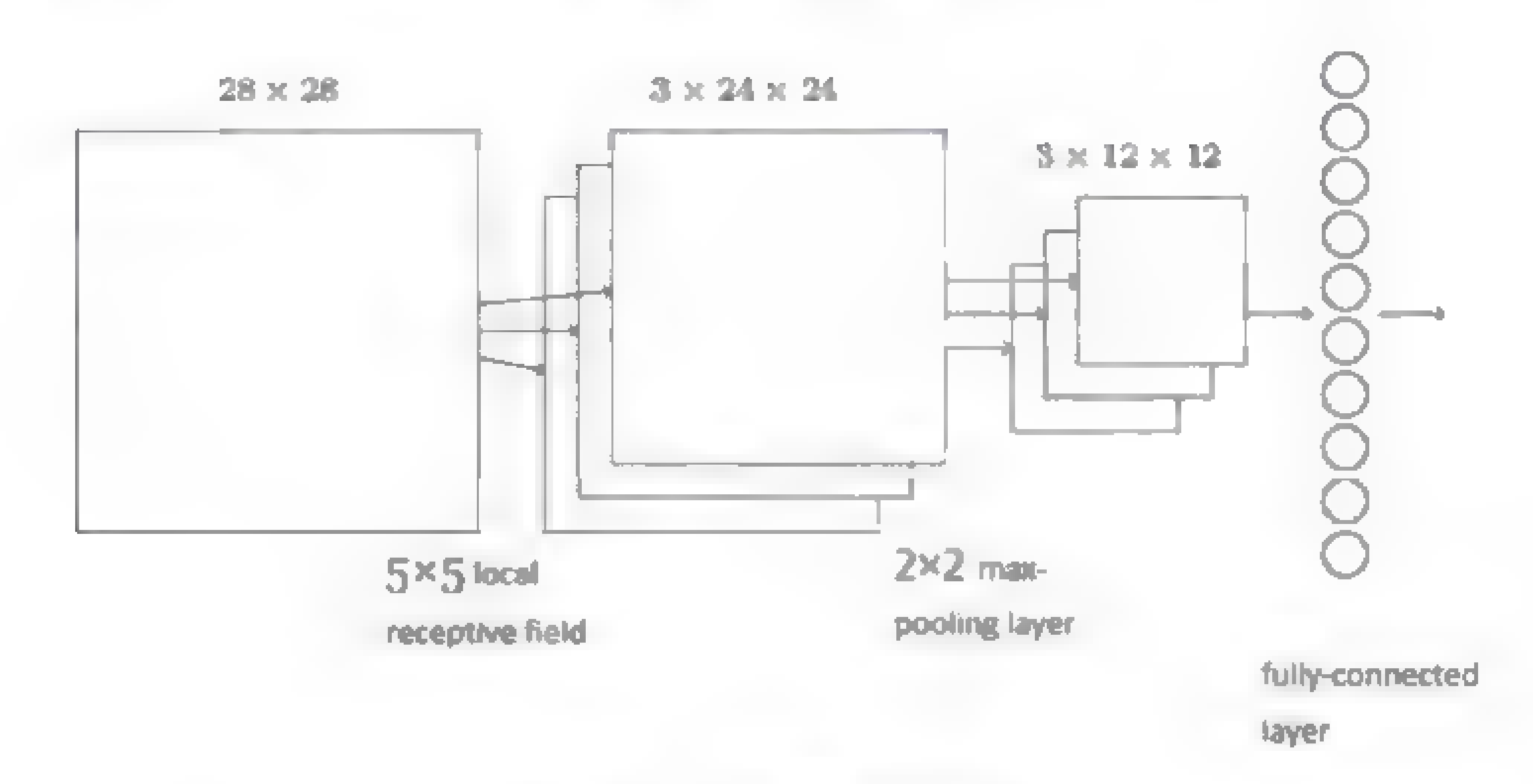


图 5-21　全连接示例图

从根本上讲，全连接层就是高度提纯的输入特征，通过最后一层的 softmax 激活函数进行分类或回归。

5.7　整合各层并使用反向传播进行训练

如上所述，卷积和池化层充当输入图像的特征提取器，而完全连接层充当分类器。

下面以图 5-22 训练 ConvNet 为例，由于输入图像是船，因此对于船（Boat）这一类的目标概率为 1，对于其他三个类的目标概率为 0，即：

- 输入图像=船
- 目标向量= [0,0,1,0]

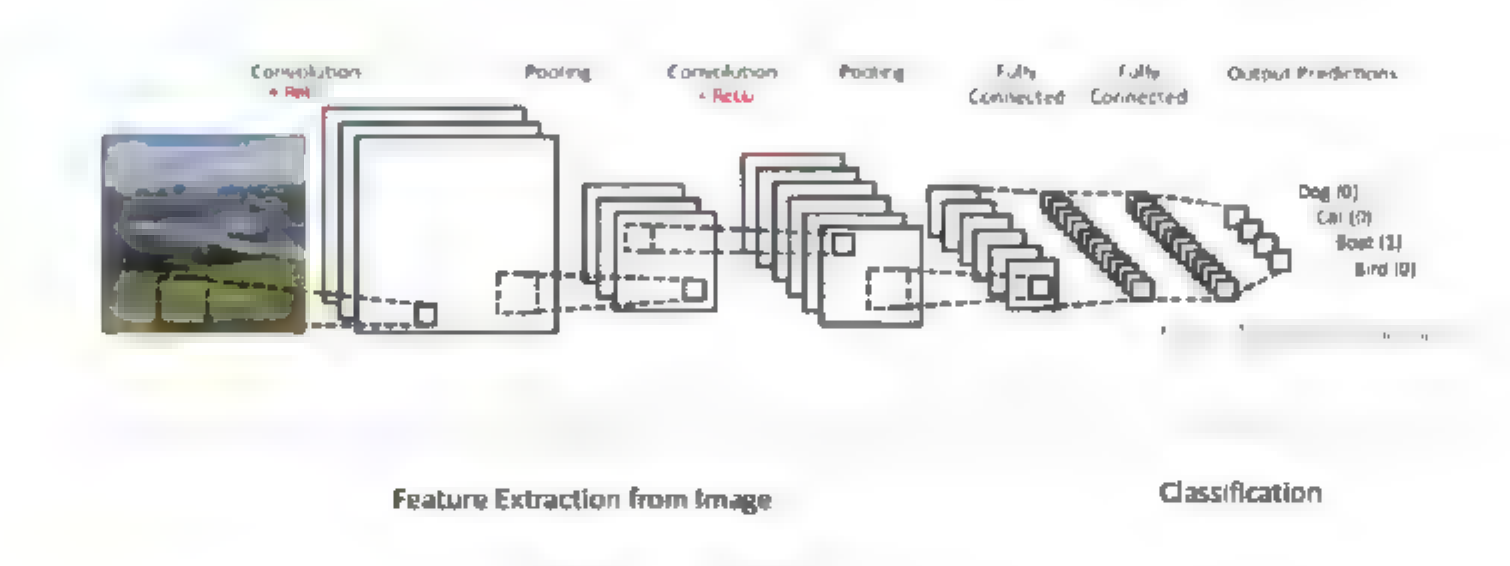


图 5-22　训练 ConvNet

卷积神经网络整体训练过程如下：

（1）使用随机值初始化所有过滤器和参数/权重值。

（2）网络将训练图像作为输入，经过前向传播步骤（卷积、ReLU 和池化运算以及完全连接层中的前向传播），并找到每个类的输出概率。

可以说上面船形图的输出概率是[0.2,0.4,0.1,0.3]。

由于在第一个训练示例中权重值是被随机分配的，因此输出概率也是随机的。

（3）计算输出层的总误差（所有 4 个类的总和）。

总误差=Σ½（目标概率 - 输出概率）²

（4）利用反向传播算法，根据网络中所有权重值计算误差的梯度，并使用梯度下降来更新所有过滤器值/权重值和参数值，以最大限度地减少输出误差。

- 权重值的调整与它们对总误差的贡献成比例。
- 当再次输入相同的图像时，输出概率现在可能是[0.1,0.1,0.7,0.1]，更接近目标向量[0,0,1,0]。
- 这意味着网络已经学会通过调整其权重值/过滤器来正确地对该具体图像进行分类，从而减少输出误差。
- 过滤器数量、过滤器大小、网络架构等参数都已在步骤（1）之前得到修复固定，并且在训练过程中不会发生变化，只更新过滤器矩阵和连接权重值。

（5）对训练集中的所有图像重复步骤（2）~（4）。

利用上述步骤训练 ConvNet，实际上意味着 ConvNet 的所有权重值和参数已经过优化，可以正确分类来自训练集的图像。

当一个新的（未知的）图像被输入到 ConvNet 中时，图中网络将经历前向传播步骤并输出每个类别的概率（对于新图像，输出概率使用已经优化的权重值来计算，以便正确分类所有之前的训练样例）。如果我们的训练样本足够大，网络将有机会对新图像有很好的泛化作用，并把它们分到正确的类别中去。

5.8 常见经典卷积神经网络

5.8.1 AlexNet

1. AlexNet 简述

AlexNet 是由 Hinton 和他的学生 Alex Krizhevsky 等人（论文署名是 Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E. Hinton，这不是重点，读者了解一下即可）于 2012 年在其经典论文《*ImageNet Classification with Deep Convolutional Neural Networks*》中提出的深度卷积网络，可以看作是 Lenet 的一种加深加宽版本，同时该论文也是 CNN 领域的经典之作。对于论文中提出的 AlexNet 网络，他们采用了一系列新的技术构件：成功应用了 ReLU、Dropout 和 LRN 等策略，第一次使用了 GPU 来进行加速，并且作者还开源了他们在 GPU 上训练网络的 CUDA 代码。AlexNet 的网络结构如图 5-23 所示，整个网络包含了六亿三千万个连接、六千万个参数和六十五万个神经元，包括 5 个卷积层（图中的 C1、C2、C3、C4、C5），其中的三个后面还连接了最大池化层，最后还用了 3 个全连接层（图中的 R1、R2、R3）。在 2012 年，AlexNet 以绝对的优势赢得了当年 ILSVRC 的比赛

冠军，top-5 错误率降低至 16.4%，相比于第二名 26.2%的成绩有了巨大的改善。正是由于 AlexNet 的出现，处于低谷期的神经网络被再次推到世人面前，并将深度学习（深度卷积网络）在计算及视觉领域置于统治地位，同时加速了深度学习在语音识别、自然语言处理等领域的拓展应用。

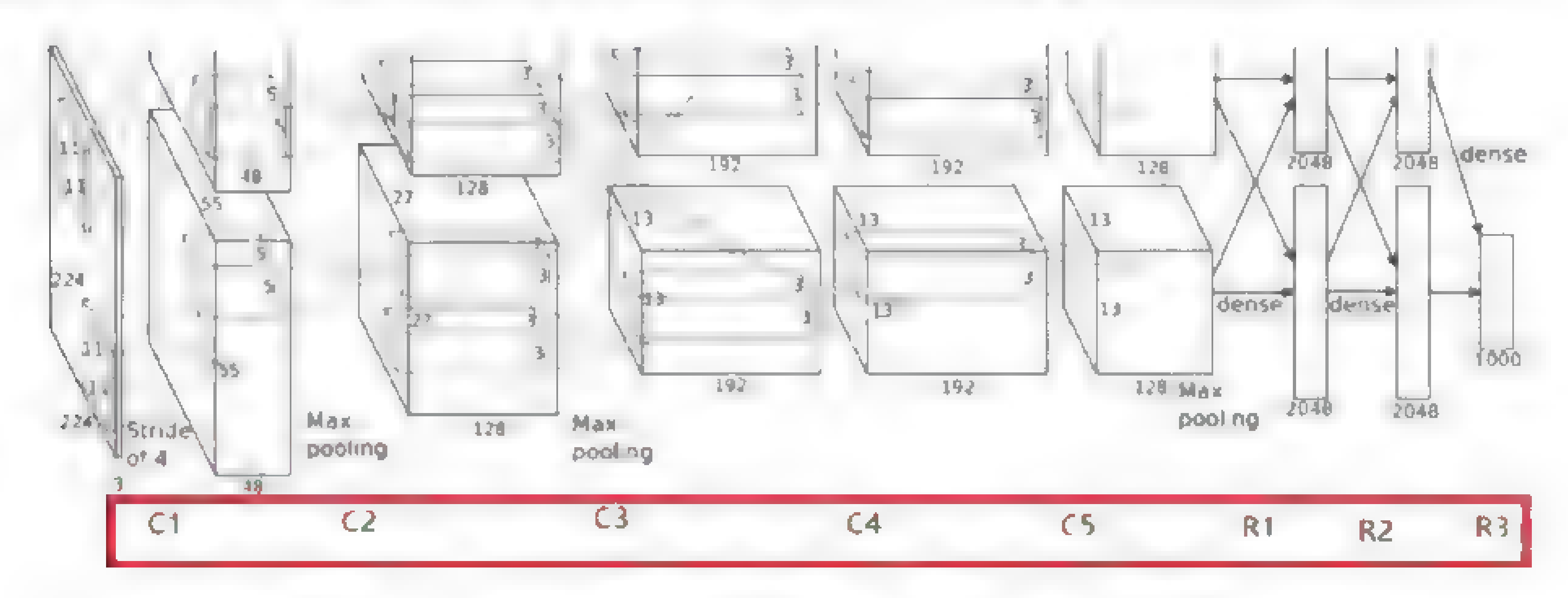


图 5-23　AlexNet 网络结构示意图（在两个 GPU 下进行）

2. AlexNet 基本结构详细解析

在 AlexNet 中，对于每次卷积后的结果，我们都可以在图 5-23 中看到。例如，经过卷积层 C1 后，原始的图像变成了 55×55 的尺寸，一共有 96 个通道，分布在 2 张 3GB 的显卡上，所以图 5-23 中一个立方体的尺寸是 55×55×48（48 是通道数目），而在这个立方体里面还有一个 5×5×48 的小立方体，这个就是 C2 卷积层的核尺寸，48 是核的厚度。这样我们就能看到每一层的卷积核尺寸以及每一层卷积之后的尺寸了。

> **注意**
>
> 大多数情况下，我们看到的 AlexNet 网络结构都是用图 5-23 来表示的，但是实际上，输入图像的尺寸不是 224×224×3，而是 227×227×3（其实可以用 224 的尺寸推导一下，就会发现边界填充的结果是小数，这明显是不对的。

下面我们对 ALexNet 网络结构进行详细解析。

（1）第一层输入数据为原始的 227×227×3 的图像，这个图像被 11×11×3 的卷积核进行卷积运算，卷积核对原始图像的每次卷积都生成一个新的像素。卷积核沿原始图像的 x 轴方向和 y 轴方向移动，移动的步长是 4 个像素。因此，卷积核在移动的过程中会生成(227-11)÷4+1=55 个像素，行和列的 55×55 个像素形成对原始图像卷积之后的像素层。总计有 96 个卷积核，会生成 55×55×96 个卷积后的像素层。96 个卷积核分成 2 组，每组 48 个卷积核，则对应生成 2 组 55×55×48 的卷积后的像素层数据。这些像素层经过相关处理后生成激活像素层，尺寸仍为 2 组 55×55×48 的像素层数据。

这些像素层经过池化运算处理（池化运算的尺度为 3×3，运算的步长为 2），则池化后图像的尺寸为(55-3)÷2+1=27。即池化后像素的规模为 27×27×96；然后经过归一化处理，归一化运算的尺度为 5×5；第一卷积层运算结束后形成的像素层的规模为 27×27×96。分别对应 96 个卷积核所运算形成。这 96 层像素层分为 2 组，每组 48 个像素层，每组在一个独立的 GPU 上进行运算。

反向传播时，每个卷积核对应一个偏差值，即第一层的 96 个卷积核对应上层输入的 96 个偏差值。

（2）第二层输入数据为第一层输出的 $27\times27\times96$ 的像素层，为便于后续处理，每幅像素层的左右两边和上下两边都要填充 2 个像素；$27\times27\times96$ 的像素数据分成 $27\times27\times48$ 的两组像素数据，两组数据分别在两个不同的 GPU 中进行运算。每组像素数据被 $5\times5\times48$ 的卷积核进行卷积运算，卷积核对每组数据的每次卷积都生成一个新的像素。卷积核沿原始图像的 x 轴方向和 y 轴方向移动，移动的步长是 1 个像素。因此，卷积核在移动的过程中会生成 $(27-5+2\times2)/1+1=27$ 个像素（27 个像素减去 5，正好是 22，加上上下、左右各填充的 2 个像素，即生成 26 个像素，再加上被减去的 5 也对应生成一个像素）。行和列的 27×27 个像素形成对原始图像卷积之后的像素层。共有 256 个 $5\times5\times48$ 卷积核；这 256 个卷积核分成两组，每组针对一个 GPU 中的 $27\times27\times48$ 的像素进行卷积运算，会生成两组 $27\times27\times128$ 个卷积后的像素层。这些像素层经过相关处理后生成激活像素层，尺寸仍为两组 $27\times27\times128$ 的像素层。

这些像素层经过池化运算的处理（池化运算的尺度为 3×3，运算的步长为 2），则池化后图像的尺寸为 $(57-3)\div2+1=13$，即池化后像素的规模为 2 组 $13\times13\times128$ 的像素层；然后经过归一化处理，归一化运算的尺度为 5×5；第二卷积层运算结束后形成的像素层的规模为 2 组 $13\times13\times128$ 的像素层，分别对应 2 组 128 个卷积核运算形成。每组在一个 GPU 上进行运算，即共 256 个卷积核，共 2 个 GPU 进行运算。

反向传播时，每个卷积核对应一个偏差值，即第一层的 96 个卷积核对应上层输入的 256 个偏差值。

（3）第三层输入数据为第二层输出的 2 组 $13\times13\times128$ 的像素层，为便于后续处理，每幅像素层的左右两边和上下两边都要填充 1 个像素；2 组像素层数据都被送至 2 个不同的 GPU 中进行运算。每个 GPU 中都有 192 个卷积核，每个卷积核的尺寸是 $3\times3\times256$。因此，每个 GPU 中的卷积核都能对 2 组 $13\times13\times128$ 的像素层的所有数据进行卷积运算。卷积核对每组数据的每次卷积都生成一个新的像素。卷积核沿像素层数据的 x 轴方向和 y 轴方向移动，移动的步长是 1 个像素。因此，运算后的卷积核的尺寸为 $(13-3+1\times2)\div1+1=13$（13 个像素减去 3，正好是 10，加上上下、左右各填充的 1 个像素，即生成 12 个像素，再加上被减去的 3 也对应生成一个像素），每个 GPU 中共 $13\times13\times192$ 个卷积核。2 个 GPU 中共 $13\times13\times384$ 个卷积后的像素层。同样，这些像素层经过处理后，生成激活像素层，尺寸仍为 2 组 $13\times13\times192$ 像素层，共 $13\times13\times384$ 个像素层。

（4）第四层输入数据为第三层输出的 2 组 $13\times13\times192$ 的像素层，为便于后续处理，每幅像素层的左右两边和上下两边都要填充 1 个像素；2 组像素层数据都被送至 2 个不同的 GPU 中进行运算。每个 GPU 中都有 192 个卷积核，每个卷积核的尺寸是 $3\times3\times192$。因此，每个 GPU 中的卷积核能对 1 组 $13\times13\times192$ 的像素层的数据进行卷积运算。卷积核对每组数据的每次卷积都生成一个新的像素。卷积核沿像素层数据的 x 轴方向和 y 轴方向移动，移动的步长是 1 个像素。因此，运算后的卷积核的尺寸为 $(13-3+1\times2)\div1+1=13$（13 个像素减去 3，正好是 10，加上上下、左右各填充的 1 个像素，即生成 12 个像素，再加上被减去的 3 也对应生成一个像素），每个 GPU 中共 $13\times13\times192$ 个卷积核。2 个 GPU 中共 $13\times13\times384$ 个卷积后的像素层。同样，这些像素层经过处理后生成激活像素层，尺寸仍为 2 组 $13\times13\times192$ 像素层，共 $13\times13\times384$ 个像素层。

（5）第五层输入数据为第四层输出的 2 组 13×13×192 的像素层，为便于后续处理，每幅像素层的左右两边和上下两边都要填充 1 个像素；2 组像素层数据都被送至 2 个不同的 GPU 中进行运算。每个 GPU 中都有 128 个卷积核，每个卷积核的尺寸是 3×3×192。因此，每个 GPU 中的卷积核能对 1 组 13×13×192 的像素层的数据进行卷积运算。卷积核对每组数据的每次卷积都生成一个新的像素。卷积核沿像素层数据的 x 轴方向和 y 轴方向移动，移动的步长是 1 个像素。因此，运算后的卷积核的尺寸为(13-3+1×2)÷1+1=13（13 个像素减去 3，正好是 10，加上上下、左右各填充的 1 个像素，即生成 12 个像素，再加上被减去的 3 也对应生成一个像素），每个 GPU 中共 13×13×128 个卷积核。2 个 GPU 中共 13×13×256 个卷积后的像素层。这些像素层经过处理后生成激活像素层，尺寸仍为 2 组 13×13×128 像素层，共 13×13×256 个像素层。

2 组 13×13×128 像素层分别在 2 个不同 GPU 中进行池化运算处理。池化运算的尺度为 3×3，运算的步长为 2，池化后图像的尺寸为(13-3)÷2+1=6，即池化后像素的规模为两组 6×6×128 的像素层数据，共 6×6×256 个像素层数据。

（6）第六层输入数据的尺寸是 6×6×256，采用 6×6×256 尺寸的过滤器对第六层的输入数据进行卷积运算；每个 6×6×256 尺寸的过滤器对第六层的输入数据进行卷积运算生成一个运算结果，通过一个神经元输出这个运算结果；共有 4096 个 6×6×256 尺寸的过滤器对输入数据进行卷积运算，通过 4096 个神经元输出运算结果；这 4096 个运算结果通过 ReLU 激活函数生成 4096 个值；并通过 drop 运算后输出 4096 个本层的输出结果值。

由于第六层的运算过程中采用的过滤器的尺寸(6×6×256)与待处理的特征图（Feature Map）的尺寸(6×6×256)相同，即过滤器中的每个系数只与特征图中的一个像素值相乘；而其他卷积层中，每个过滤器的系数都会与多个特征图中像素值相乘；因此，将第六层称为全连接层。

第五层输出的 6×6×256 规模的像素层数据与第六层的 4096 个神经元进行全连接，然后经由 relu6 进行处理后生成 4096 个数据，再经过 dropout6 处理后输出 4096 个数据。

（7）第七层输出的 4096 个数据与第八层的 1000 个神经元进行全连接，经过训练后输出被训练的数值。

3. AlexNet 网络优势解析

前面讲了一下 AlexNet 的基本网络结构，大家肯定会对其中的一些点产生疑问，比如 LRN、ReLU、dropout，相信接触过深度学习的读者都听说或者了解过这些。这里将详细讲述这些东西为什么能提高最终网络的性能。

（1）ReLU 非线性

大家可能都知道，在浅层的神经网络中，引入非线性操作（也就是通常说的激活函数）能够提高神经网络的泛化能力，进而使得神经网络更具有鲁棒性。但是在深层网络中，如果我们选择 tanh 函数作为激活函数，其实会使得模型的计算量增加、训练速度变慢。Hinton 等在论文中引入 ReLU（Rectified Linear Units）来作为激活函数：

$$Relu(x) = max(0,x)$$

实际上，基于 ReLU 的深度卷积网络比基于 tanh 的网络在训练上会提高数倍。图 5-24 给出了

一个基于 CIFAR-10 的四层卷积网络在 tanh 和 ReLU 达到 25%的训练错误率（Training Error）时的迭代次数。

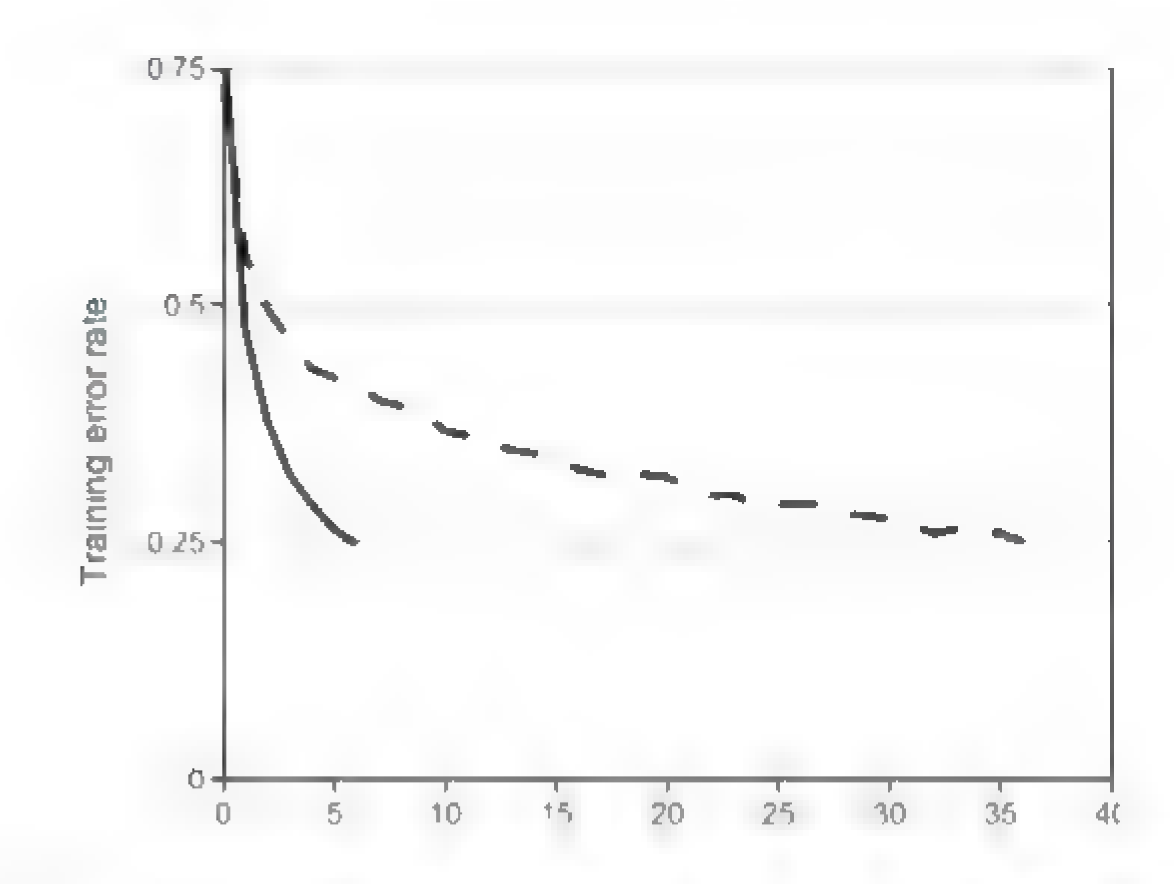


图 5-24　tanh 和 ReLU 达到 25%的训练错误率（Training Error）随着迭代次数的变化情况

在图 5-24 中，实线、虚线分别代表的是 ReLU、tanh 的训练错误率的情况，可以发现，ReLU 比 tanh 收敛更快。

之所以会出现图 5-24 中的现象，我们可以从以下两个方面的原因去思考：其一，简单的 max 计算可以大幅度减少计算量，进而提升训练速度（论文中提到过）；其二，在 ReLU 中，梯度是被直接传递的，而我们知道在深度网络中梯度是存在衰减的，而 ReLU 却能够最大限度地维持梯度，使得梯度衰减的趋势得以减缓。此外，由于反向传播过程中没有了梯度换算的操作，因此可以加快训练的进程。

（2）局部响应归一化（Local Response Normalization，LRU）

在这里，我们为了不让某一些核（Kernel）的权重值（Weight）变得很大，就需要对不同的核进行归一化（Normalization）处理。之所以这样，是因为：如果某一个核的权重值变得很大，那么它的权重值只要发生略微的变化就会对网络性能产生很大的影响，这一点可以结合神经学中的侧抑制（Lateral Inhibition）概念来理解，即活跃的神经元会对与它相邻的神经元产生抑制作用。

$$b_{(x,y)}^{i}=\frac{a_{(x,y)}^{i}}{\left(k+\alpha\sum_{\left(j=max\left(0,i-\frac{n}{2}\right)\right)}^{min\left(N-1,i+\frac{n}{2}\right)}\left(a_{(x,y)}^{i}\right)^{2}\right)^{\beta}}$$

其中，这里的k、n、α、β常量都是“可调参数”，由最好的验证集（Validation Set）来决定。

（3）Dropout

这是一种正则化手段，能够比较有效地防止神经网络的过拟合（Overfitting），通过随机地将部分神经元的输出置零来实现。

相对于一般（如线性模型）使用正则的方法来防止模型过拟合，在神经网络中 Dropout 通过修改神经网络本身的结构来实现。在全连接层，将某些隐藏层神经元的输出置为 0，对于每个神经元输出置为 0 的概率是 0.5，Dropout 的神经元不会对前向传播操作造成影响，也退出了反向传导权

重值修正。这样在提高训练效率同时也防止了过拟合现象的发生。论文中梯度神经网络在前面的两个全连接层进行了 Dropout，有效地阻止了过拟合现象。

（4）数据扩充（Data Augmentation）

在深度学习中，当数据量不够大的时候，采取的方法之一便是数据扩充（其他还有 Regularization、Dropout 等）：通过平移、翻转、加噪声等方法从已有数据中创造出一批“新”的数据。通过数据扩充，我们可以防止模型出现过拟合现象，进而提升模型的性能。在 Hinton 和 Alex Krizhevsky 的论文中，是从 256×256 中随机提出 227×227 的 patches（论文中是 224×224），还有就是通过 PCA 来扩充数据集。这样就很有效地扩充了数据集，其实还有更多的方法，可以根据我们的业务场景去选择使用，比如进行基本的图像转换（如增加/减少亮度、采用一些滤光算法等），这是一种特别有效的手段，尤其是当数据量不够大的时候。

5.8.2　VGGNet

1. VGGNet 简述

VGGNet 是于 2014 年由牛津大学计算机视觉组（Visual Geometry Group）和 Google DeepMind 公司研究员一起研发的新型深度卷积神经网络。VGGNet 探索了卷积神经网络的深度与其性能之间的关系，通过反复堆叠 3×3 的小型卷积核和 2×2 的最大池化层，VGGNet 成功地构筑了 16~19 层深的卷积神经网络。VGGNet 证明了增加网络的深度能够在一定程度上影响网络最终的性能，相比之前的网络结构，使错误率大幅下降，并取得了 ILSVRC2014 比赛分类项目的第二名（第一名是 GoogLeNet，也是同年提出的）和定位项目的第一名。同时 VGGNet 的拓展性很强，迁移到其他图片数据上的泛化性也非常好。VGGNet 的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（3×3）和最大池化尺寸（2×2）。目前，VGGNet 依然经常被用来提取图像特征。VGGNet 训练后的模型参数在其官方网站已经开源了，可用来在特定领域的图像分类任务上进行再训练（相当于提供了非常好的初始化权重值），因此被广泛采用。

2. VGGNet 结构详解

在 VGGNet 论文《*Very Deep Convolutional Networks for Large-Scale Image Recognition*》（基于深层卷积网络的大规模图像识别）中，作者全部使用 3×3 的卷积核和 2×2 的池化核，借助不断加深的网络结构来提高模型性能。图 5-25 给出了 VGGNet 中各级别的网络结构示意图。论文作者分别使用了 A、A-LRN、B、C、D、E 这 6 种网络结构进行测试，这 6 种网络结构均相似，都是由 5 层卷积层、3 层全连接层组成，其中的区别在于每个卷积层的子层数量不同，从 A 至 E 依次增加（子层数量从 1 到 4），总的网络深度从 11 层到 19 层（添加的层以粗体显示），图中的卷积层参数表示为“conv 感受野大小-通道数”，例如 con3-128，表示使用 3×3 的卷积核，通道数为 128。为了简洁起见，在表格中不显示 ReLU 激活功能。

VGG16 VGG19

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 5-25 VGGNet 各级别的网络参数结构示意图

图 5-26 给出了每一级别的参数个数，从第 11 层的网络一直到第 19 层的网络都有详尽的性能测试。虽然从 A 到 E 每一级网络都逐渐加深，但是网络的参数个数并没有增加很多，这是因为参数个数大部分都消耗在最后 3 个全连接层上了。前面的卷积部分虽然很深，但是消失的参数个数并不大，而训练比较耗时的部分仍然是卷积部分（该部分计算量较大）。这其中的网络结构 D 和网络结构 E 就是我们常说的 VGG16 和 VGG19。

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

图 5-26 VGGNet 各级别网络参数个数（单位：百万）

接着，我们以 VGG16（网络结构 D）为例介绍其各层的处理过程，见图 5-27，A、A-LRN、B、C、E 等其他网络结构的处理过程与网络结构 D 类似。其处理过程如下（请对比图 5-25、图 5-26 和图 5-27 的数字变化，有助于理解 VGG16 的处理过程）：

（1）输入 224×224×3 的图片，经 64 个 3×3 的卷积核做两次卷积+ReLU，卷积后的尺寸变为 224×224×64。

（2）做最大池化（Max Pooling），池化单元尺寸为 2×2（效果为图像尺寸减半），池化后的尺寸变为 112×112×64。

（3）经 128 个 3×3 的卷积核做两次卷积+ReLU，尺寸变为 112×112×128。

（4）做 2×2 的最大池化，尺寸变为 56×56×128。

（5）经 256 个 3×3 的卷积核做三次卷积+ReLU，尺寸变为 56×56×256。

（6）做 2×2 的最大池化，尺寸变为 28×28×256。

（7）经 512 个 3×3 的卷积核做三次卷积+ReLU，尺寸变为 28×28×512。

（8）做 2×2 的最大池化，尺寸变为 14×14×512。

（9）经 512 个 3×3 的卷积核做三次卷积+ReLU，尺寸变为 14×14×512。

（10）做 2×2 的最大池化，尺寸变为 7×7×512。

（11）与两层 1×1×4096、一层 1×1×1000 进行全连接+ReLU（共三层）。

（12）通过 softmax 输出 1000 个预测结果。

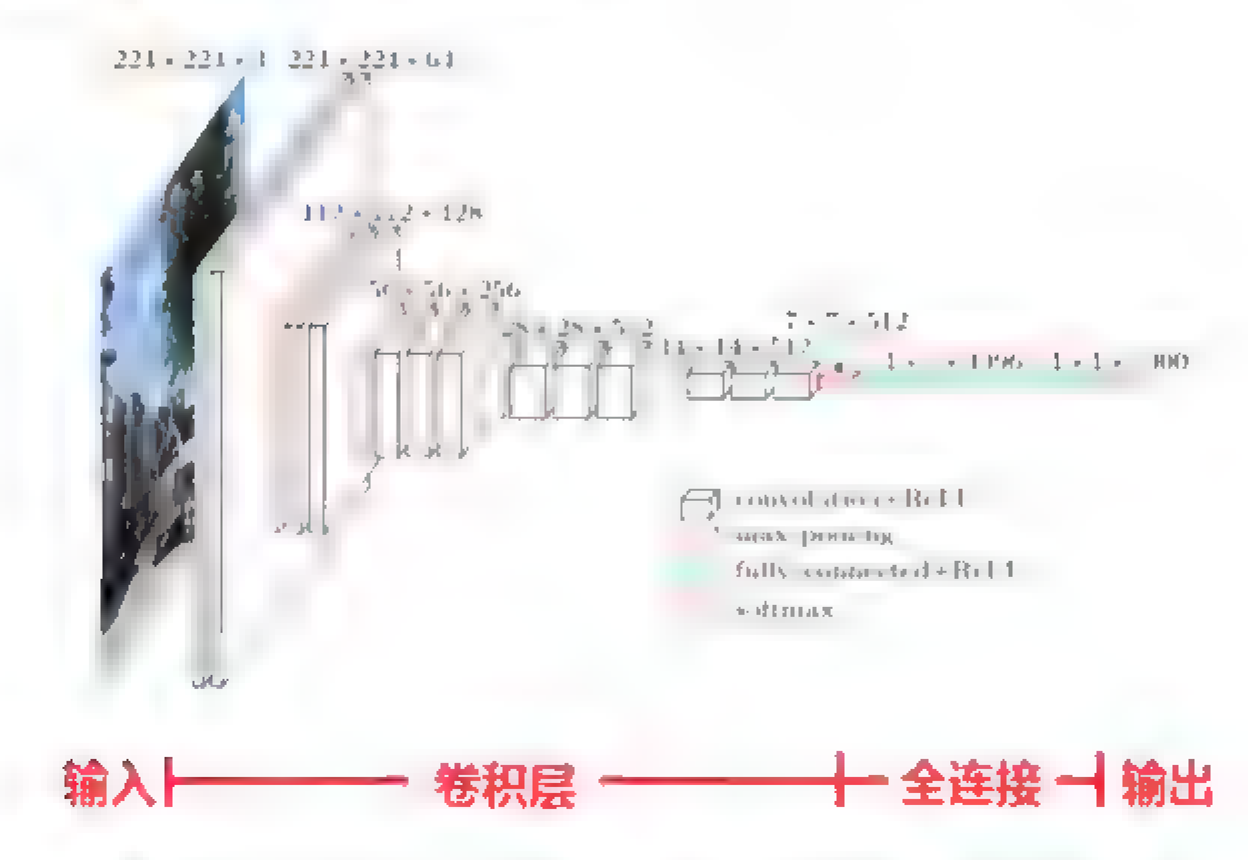


图 5-27　VGG16（网络结构 D）各层处理过程示意图

图 5-28 给出了 VGGNet 使用 Multi-Scale 训练时得到的结果，可以发现网络结构 D 和 E 均能够达到 7.5%的错误率。在对比 VGGNet 各级别网络后得出以下 3 个观点：

（1）LRN 层作用不大。

（2）越深层的网络效果越佳。

（3）2×2 的卷积也是有效的，但没有 3×3 的卷积好，即大一些的卷积核能够学习更大的空间特征。

ConvNet config. (Table 1)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
B	256	224,256,288	28.2	9.6
C	256	224,256,288	27.7	9.2
	384	352,384,416	27.8	9.2
	[256; 512]	256,384,512	26.3	8.2
D	256	224,256,288	26.6	8.6
	384	352,384,416	26.5	8.6
	[256; 512]	256,384,512	**24.8**	**7.5**
E	256	224,256,288	26.9	8.7
	384	352,384,416	26.7	8.6
	[256; 512]	256,384,512	**24.8**	**7.5**

图 5-28　VGGNet 各级别在使用 Multi-Scale 训练时的 top-5 错误率

5.8.3 Google Inception Net

Google Inception Net（有时也称 GoogLeNet）和 VGGNet 是 2014 年 ImageNet 挑战赛（ILSVRC14）的翘楚，分别获得了第一名和第二名，这两类模型结构的共同特点是神经网络层次更深了。其中 VGGNet 继承了 LeNet 以及 AlexNet 的一些框架结构，而 GoogLeNet 则做了更加大胆的网络结构尝试。2014 年这届比赛中的 Inception Net 一般被称为 Inception V1，其中一个很重要的特点是控制了计算量和参数量的同时取得了非常出色的分类性能：top-5 错误率为 6.67%，不到 AlexNet 的一半。Google Inception Net 其实是一个大的家族，成员有：

（1）2014 年 9 月，论文《*Going Deeper with Convolutions*》提出的 Inception V1（top-5 错误率为 6.67%）。

（2）2015 年 2 月，论文《*Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*》提出的 Inception V2（top-5 错误率为 4.8%）。

（3）2015 年 12 月，论文《*Rethinking the Inception Architecture for Computer Vision*》提出的 Inception V3（top-5 错误率 3.5%）。

（4）2016 年 2 月，论文《*Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*》提出的 Inception V4（top-5 错误率 3.08%）。

1. Inception V1

Inception V1 虽然深度只有 22 层，但大小却比 AlexNet 和 VGGNet 小很多，GoogleNet 参数为 500 万个，仅为 AlexNet 参数个数的 1/12 倍，是 VGGNet 参数个数的 1/4，所以在内存或计算资源有限的时候，GoogleNet 是比较好的选择；从模型的效果来看，GoogleNet 的性能也是更佳的选择。Inception V1 模型的参数量少但效果好的原因除了模型层数更深、表达能力更强外，还有如下两方面的原因：

（1）放弃了最后的全连接层，代之以全局平均池化层（将图片尺寸变为 1×1）。由于全连接层几乎占据了 AlexNet 或 VGGNet 中 90%的参数量，且这会引发过拟合现象。采用全局平均池化层代替全连接层的做法借鉴了 Network In Network（NIN）论文的思想，这样模型训练起来速度更快且也减轻了过拟合的压力。

（2）Inception V1 中精心设计的 Inception 模块（Inception Module）提升了参数的利用效率，具体的网络结构见图 5-29。这一部分借鉴了 NIN 的思想，可以理解为 Inception 模块本身就如同大网络中的一个小网络，其结构可以反复叠加在一起形成大网络。但 Inception V1 比 NIN 更具有优势的地方是它增加了分支网络，NIN 则主要是级联的卷积层和 MLPConv 层。

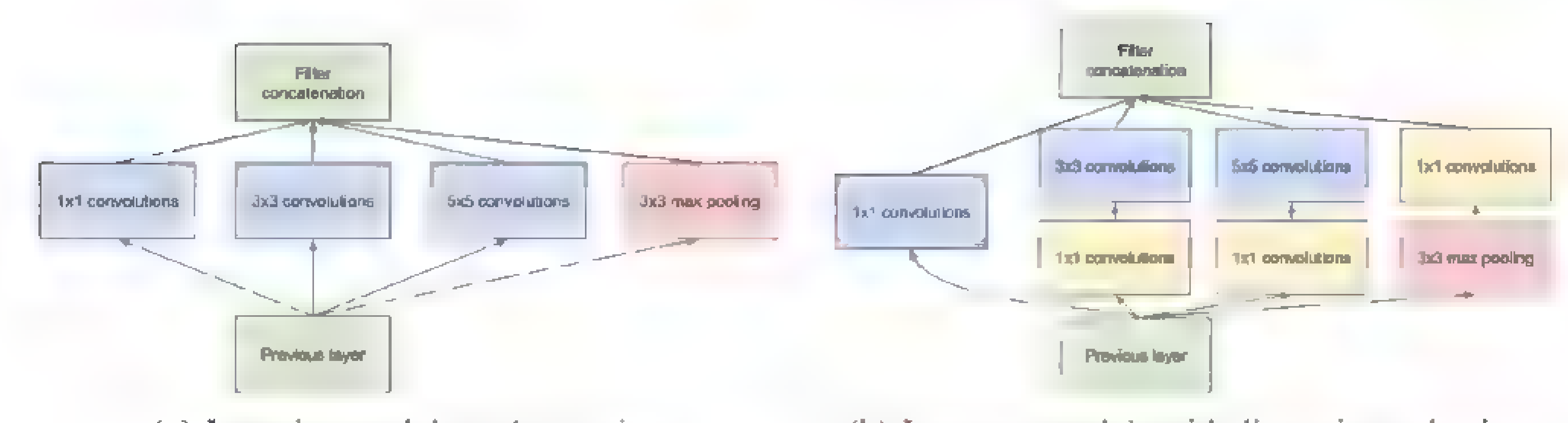


图 5-29　Inception V1 网络结构示意图（图片来自论文《*Going Deeper with Convolutions*》）

接下来，我们给出 Inception V1 模型的特点，具体如下：

（1）卷积层共有的一个功能，可以实现通道方向的降维和增维，至于是降还是增，取决于卷积层的通道数（过滤器个数），在 Inception V1 中 1×1 卷积用于降维，减少权重值大小和特征图维度。

（2）1×1 卷积特有的功能，由于 1×1 卷积只有一个参数，相当于对原始特征图做了一个尺度变换（scale）操作，并且这个尺度变换还是训练学出来的，无疑会对识别精度有提升。

（3）增加了网络的深度。

（4）增加了网络的宽度。

（5）同时使用了 1×1、3×3、5×5 的卷积，增加了网络对尺度的适应性。

2. Inception V2

Inception V2 学习了 VGGNet，使用两个 3×3 的卷积取代了 5×5 的大卷积（用于降低参数量且减少过拟合），并提出了著名的 Batch Normalization（BN）方法，其网络结构如图 5-30 所示。BN 是一个非常有效的正则化方法，可以让大型卷积网络的训练速度加快很多倍，同时收敛后的分类准确率也可以得到很大的提升。Inception V2 的优势在于以下两点：

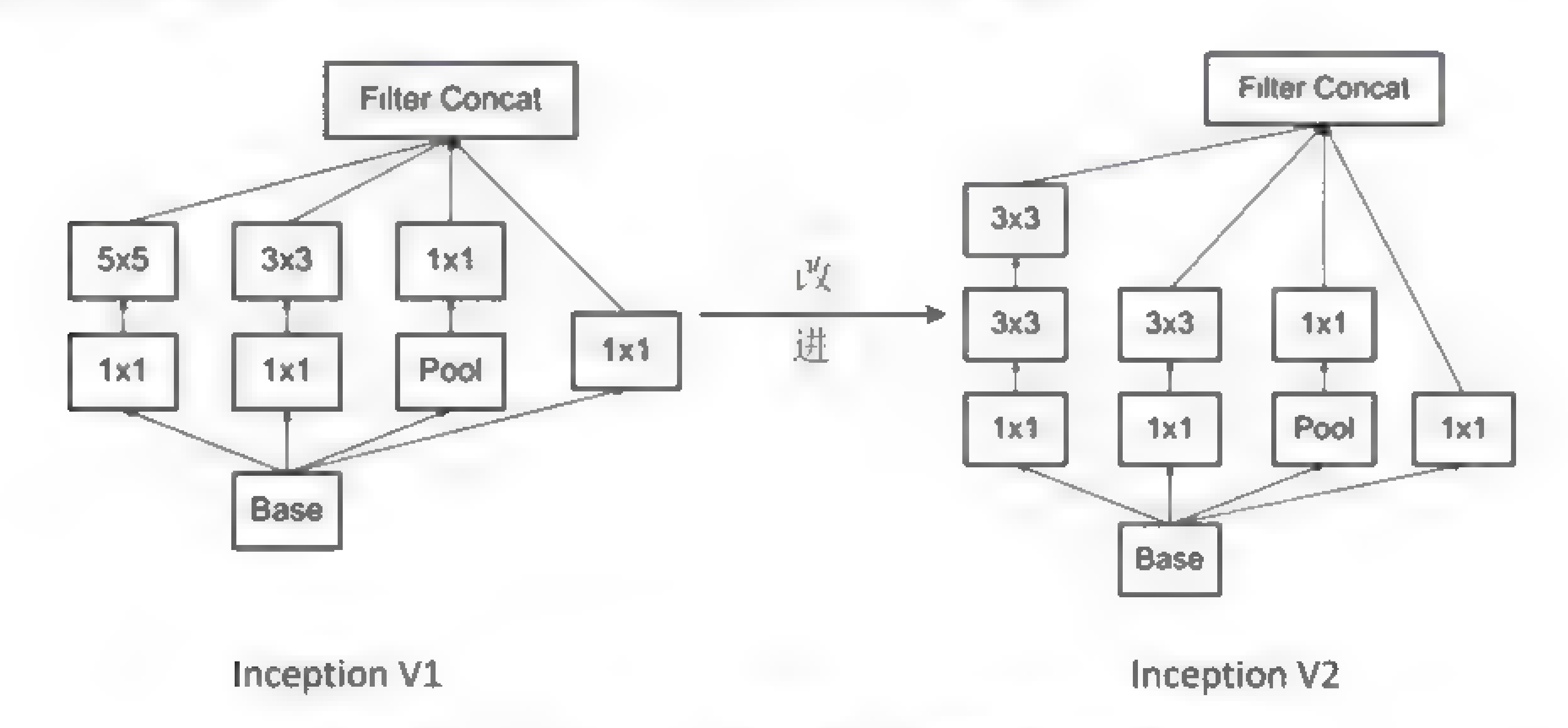


图 5-30　Inception V1 改进到 V2 的网络结构示意图

（1）加入了 BN 层，减少了内部协方差偏移（InternalCovariate Shift，内部神经元的数据分布

发生变化），使每一层的输出都规范化到一个 N(0, 1)的高斯分布（即正态分布），从而增加了模型的鲁棒性，能以更大的学习速率训练，收敛更快，初始化操作更加随意，同时作为一种正则化技术，可以减少 dropout 层的使用。

（2）用 2 个连续的 3×3 conv 替代 Inception 模块中的 5×5，从而实现网络深度的增加，网络整体深度增加了 9 层，缺点就是增加了 25%的权重值和 30%的计算量。

3. Inception V3

Inception V3 网络主要是在 V2 的基础上提出了卷积分解（Factorization），如图 5-31 所示。其主要特点有以下两点：

（1）将 7×7 分解成两个一维的卷积（1×7,7×1），3×3 也是一样（1×3,3×1），这样的好处是既可以加速计算（多余的计算能力可以用来增加网络深度），又可以将 1 个卷积分解成 2 个卷积，使得网络深度进一步增加，从而增加了网络的非线性，更加精细地设计了 35×35/17×17/8x8 的模块。

（2）增加网络宽度，网络输入从 224×224 变为了 299×299。

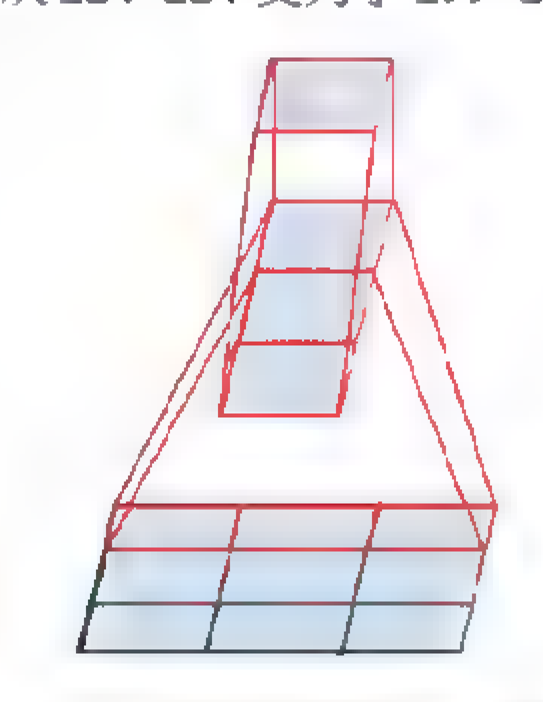


图 5-31　将一个 3×3 卷积分解成 1×3 卷积和 3×1 卷积

4. Inception V4

Inception V4 相比于 V3 主要是结合了微软的 ResNet，得到了 Inception-ResNet-v1、Inception-ResNet-v2、Inception-v4 网络。关于 ResNet 我们将在 5.8.4 节单独说明。

ResNet 的残差结构如图 5-32 所示。

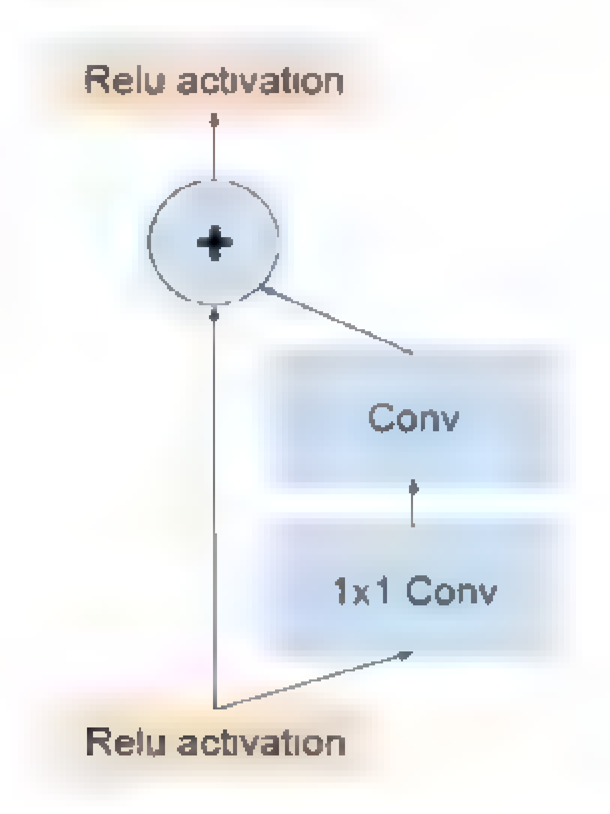


图 5-32　ResNet 的残差结构示意图

Inception 与 ResNet 结合后的网络结构如图 5-33 所示。

图 5-33　Inception 与 ResNet 结合后的网络结构示意图

现在，我们可以看一下 Inception V4 的优点，具体如下：

（1）将 Inception 模块和 ResNet 结合，提出了 Inception-ResNet-v1、Inception-ResNet-v2，使得训练加速，收敛更快，精度更高。

（2）设计了更深的 Inception-v4 版本，效果和 Inception-ResNet-v2 相当。

（3）网络输入大小和 V3 一样，还是 299×299。

5.8.4　ResNet 网络

1. 简述

ResNet（Residual Neural Network，残差神经网络）是由微软研究院的 Kaiming He、Xiangyu Zhang、Shaoqing Ren 和 Jian Sun 四名华人于 2015 年在其论文《*Deep Residual Learning for Image Recognition*》中提出的，通过使用 ResNet Unit 成功训练出了 152 层深的神经网络，并在 ILSVRC 2015 比赛中取得了冠军，在 top5 上的错误率为 3.57%，同时参数个数比 VGGNet 低，效果却非常突出。ResNet 的结构可以极快地加速超深神经网络的训练，模型准确率的提升也非常显著。正是因为 ResNet“简单与实用”并存的魅力，所以后来很多方法都是建立在 ResNet50 或者 ResNet101 基础上完成的，检测、分割、识别等领域都纷纷使用了 ResNet 网络，Alpha Zero 也使用了 ResNet，可见 ResNet 确实很受欢迎。

2. ResNet 网络结构

ResNet 主要借鉴了高速路网络（Highway Network）的思想，即允许原始输入信息直接传输到后面的网络层中（我们可以理解为，网络中增加了捷径连接或者直连通道或者说跨层连接等），但实际上 ResNet 对高速路神经网络进行了改进（残差项中原本是带有权重值的，而 ResNet 则采用恒

等映射代替之）。图 5-34 给出了 ResNet 网络两层残差学习单元的示意图。

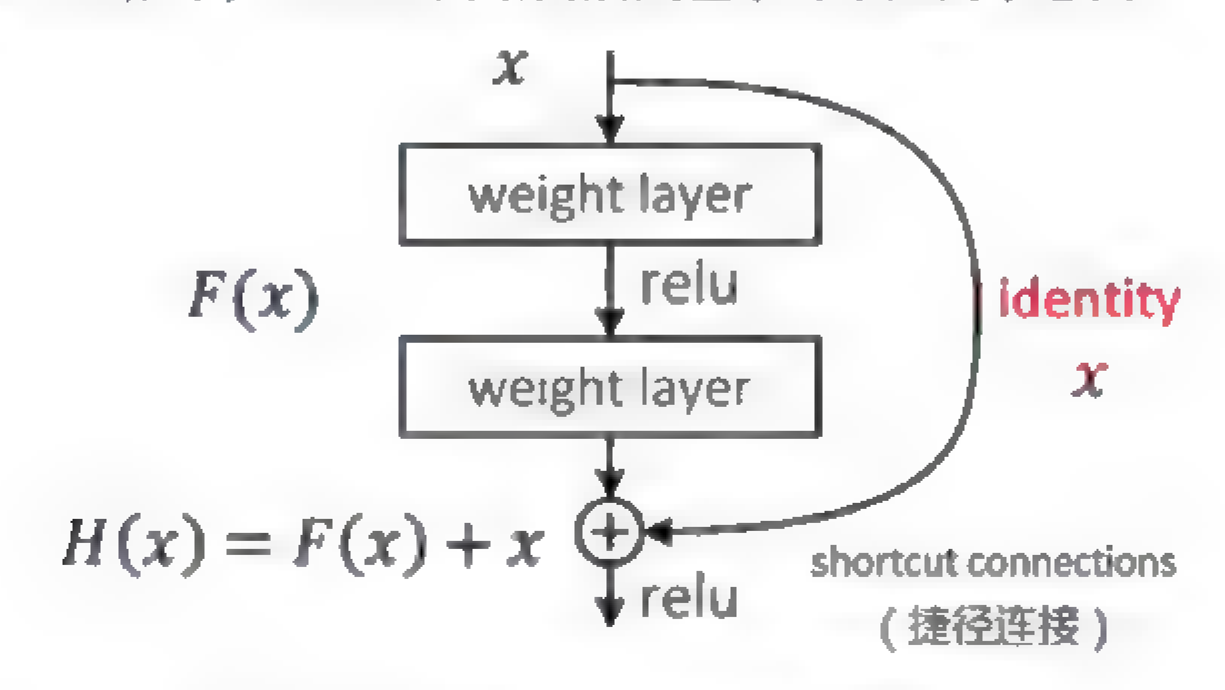


图 5-34 ResNet 网络两层残差学习单元

我们知道，在传统的卷积网络或者全连接网络当中，信息传递时或多或少会存在信息丢失、损耗等现实问题，甚至引起梯度消失或者梯度爆炸的严重问题，导致我们无法对深度神经网络进行正常训练。现在我们认识到，ResNet 可以通过其捷径连接将输入信息直接传到输出端，最大程度上保证了信息的完整性，整个神经网络只需要学习输入端和输出端存在差异的部分即可，从而大大降低了模型的学习难度。

论文《*Deep Residual Learning for Image Recognition*》中提出了 ResNet 模块的两种结构（如图 5-35 所示），其实真正在使用的 ResNet 模块并不是这么单一。这两种结构分别针对 ResNet34（左图浅层网络）和 ResNet50/101/152（右图深层网络），整个结构被称为一个“building block”。其中右图又被称为“bottleneck design”，目的就是为了降低参数的数目。我们面对的网络层数很多时，在神经网络中靠近输出端的维数就会很大，此时如果选择左侧浅层网络则会导致计算量极大，而选择右侧深层网络结构进行计算就会好很多，这里将会先用一个 1x1 卷积进行降维，然后使用一个 3x3 卷积层，最后再使用一个 1x1 卷积层恢复到之前的维度。

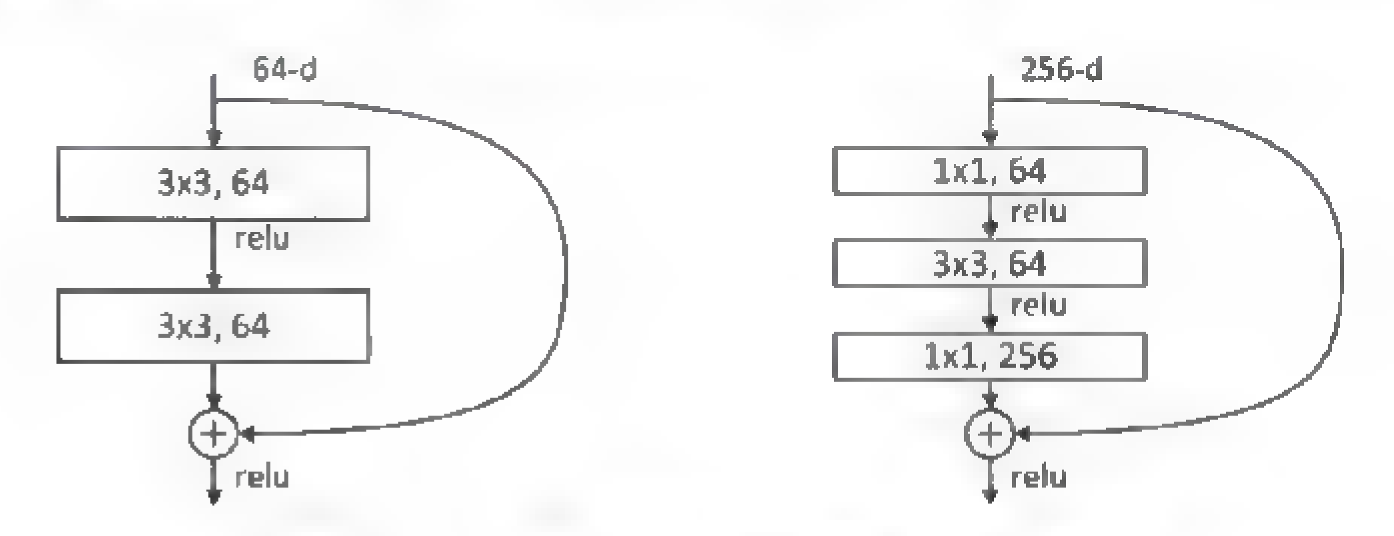


图 5-35 二层和三层的 ResNet 学习模块

5.9 利用 CNN 对 MNIST 数据集进行图片分类

本节我们分析一个在讲解卷积神经网络时经常使用到的案例，即利用 CNN 对 MNIST 数据集进行图片分类。

5.9.1　数据样本

数据样本还是计算机视觉社区中众所周知的数据集：MNIST 数据集。MNIST 数据集是从 0 到 9 的手写数字的标记图像的数据库。数据集包含三个不同的子数据集：训练、验证和测试集。下面我们将在训练集上做训练，再从测试集中随机选取一些正确和错误分类的样本，以评估 CNN 的学习能力。

5.9.2　实现 CNN

本部分的完整代码位于 ch5 文件夹的 5_cnn_image_classification_mnist.ipynb 中。

首先，我们将定义 TensorFlow 占位符，用于为输入（图像）和输出（标签）服务，然后定义一个全局步数，它被用于计算衰减的学习速率：

```
    # 输入（图像）和输出（标签）占位符
    tf_inputs = tf.placeholder(shape=[batch_size, image_size, image_size,
    n_channels],dtype=tf.float32,name='tf_mnist_images')
    tf_labels = tf.placeholder(shape=[batch_size, n_classes],
dtype=tf.float32,name='tf_mnist_labels')

    # 用于计算衰减学习速率的全局步数
    global_step = tf.Variable(0,trainable=False)
```

接下来，我们将定义模型和其他变量，它们是卷积权重值和偏差以及完全连接的权重值。

```
    # 初始化各个变量
    layer_weights = {}
    layer_biases = {}
    for layer_id in cnn_layer_ids:
        if 'pool' not in layer_id:
            layer_weights[layer_id] =tf.Variable(initial_value=
tf.random_normal(shape=layer_hyperparameters[layer_id]['weight_shape'],stddev=
0.02,dtype=tf.float32),name=layer_id+'_weights')

            layer_biases[layer_id] = tf.Variable(initial_value=tf.random_normal
(shape=[layer_hyperparameters[layer_id]['weight_shape'][-1]],
stddev=0.01,dtype=tf.float32), name=layer_id+'_bias')
```

下一步定义 logit 计算。Logits 是应用 softmax 激活之前输出层的值。为此，我们将会遍历每一层。

对于每个卷积层，将使用以下方法对输入进行卷积运算：

```
    h = tf.nn.conv2d(h,layer_weights[layer_id],layer_hyperparameters[layer_id]
['stride'],layer_hyperparameters[layer_id]['padding']) + layer_biases[layer_id]
```

这里，对于第一个卷积，输入 h 到 tf.nn.conv2d 将被替换为 tf_inputs。此外，tf.nn.conv2d(input, filter, strides, padding)按以下顺序采用有关参数值。

我们还将调用非线性激活函数 ReLU：

```
h = tf.nn.relu(h)
```

然后，对于每个池化层，使用以下方法对输入进行子采样：

```
h = tf.nn.max_pool(h, layer_hyperparameters[layer_id]['kernel_shape'],
layer_hyperparameters[layer_id]['stride'],layer_hyperparameters[layer_id]
['padding'])
```

tf.nn.max_pool（input，ksize，strides，padding）函数按顺序采用以下参数：

- input：用于子采样，形式为[batch size, input height, input width, input depth]。
- ksize：最大池化运算的每个维度上的内核大小，形式为[batch kernel size, height kernel size, width kernel size, depth kernel size]。
- strides：输入每个维度的步幅，形式为[batch stride, height stride, width stride, depth stride]。
- padding:可以是'SAME' 或'VALID'。

接下来，对于第一个完全连接的层，我们重塑输出：

```
h = tf.reshape(h,[batch_size,-1])
```

然后我们将执行权重值乘法和偏差加法，进行非线性激活：

```
h = tf.matmul(h,layer_weights[layer_id]) + layer_biases[layer_id]
h = tf.nn.relu(h)
```

这时计算 logits：

```
h = tf.matmul(h,layer_weights[layer_id]) + layer_biases[layer_id]
```

我们将 h 的最后一个值（最后一层的输出）分配给 tf_logits：

```
tf_logits = h
```

接下来，我们将定义 softmax 交叉熵损失，这是有监督分类任务的常用损失函数：

```
tf_loss = tf.nn.softmax_cross_entropy_with_logits_v2(logits=tf_logits,
labels=tf_labels)
```

我们还需要定义一个学习率，每当验证准确度没有增加预定数量的 epoch（一个 epoch 是指遍历一遍整个数据集，即训练一遍）时，我们将学习率降低 0.5 倍。这被称为学习率衰减：

```
tf_learning_rate =tf.train.exponential_decay(learning_rate=0.001,
global_step=global_step,decay_rate=0.5,decay_steps=1,staircase=True)
```

接着，我们将使用 RMSPropOptimizer 优化器定义损失最小化，有人发现它优于传统的随机梯

度下降（SGD），尤其是在计算机视觉应用中：

```
    tf_loss_minimize = tf.train.RMSPropOptimizer(learning_rate=tf_learning_rate,
momentum=0.9).minimize(tf_loss)
```

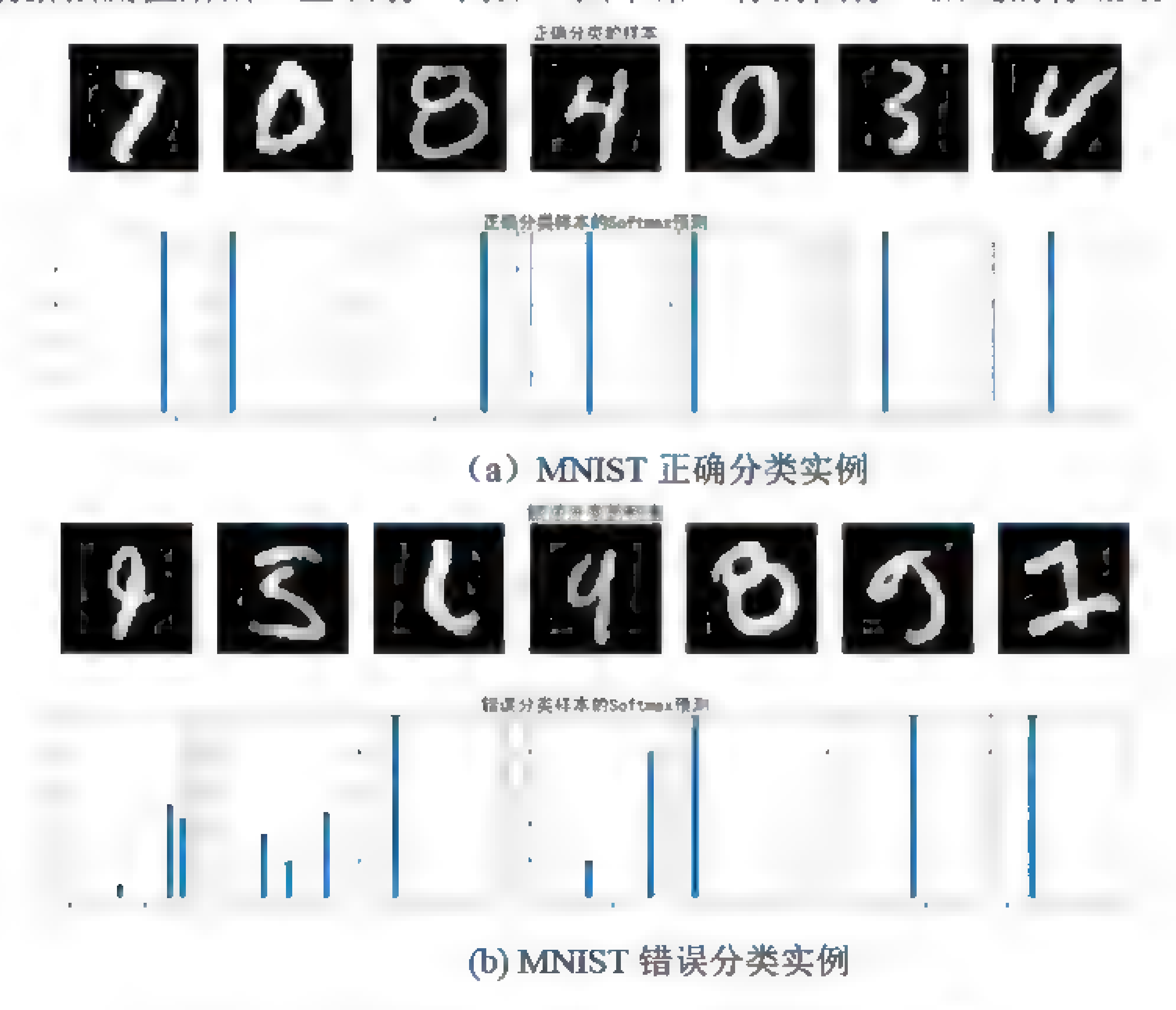
最后，通过比较预测的分类标签和实际分类标签来计算预测的准确率，我们将定义以下预测计算函数：

```
    tf_predictions = tf.nn.softmax(tf_logits)
```

目前，我们已经完成了第一个 CNN 功能的创建工作。熟悉了使用相关函数来实现 CNN 结构、定义损失、最大限度地减少损失并将预测变为未可见数据。我们使用简单的 CNN 来查看它是否可以实现对手写图像进行分类的学习。此外，通过 CNN，我们能够达到 98%以上的准确度。接下来我们将分析 CNN 产生的一些结果，将看到为什么 CNN 无法正确识别某些图像。

5.9.3　分析 CNN 产生的预测结果

在这里，我们可以从测试集中随机选取一些正确和错误分类的样本，以评估 CNN 的学习能力（见图 5-36）。我们可以看到，对于正确分类的实例，CNN 对输出非常有信心，这可以被视为学习算法的良好属性。然而，当我们评估错误分类的例子时，我们可以看到它们实际上是困难的，甚至人类也可能会弄错它们（例如，第二行从左侧起第 3 个图像），这么来看，面对不正确的样本，CNN 通常不像分类正确的样本那么自信。此外，尽管出现这些不足，但正确的标签通常不会被完全忽略，并且根据预测值给出一些识别，例如，图中第一行的图像，正确的标签基本不会被弄错。



（a）MNIST 正确分类实例

(b) MNIST 错误分类实例

图 5-36　MNIST 正确分类(a)和错误分类(b)的实例

5.10 利用 CNN 进行句子分类

虽然 CNN 主要用于计算机视觉任务，但在 NLP 领域也经常被使用到，它对处理句子分类任务是很有效的。

在句子分类任务中，把给定句子归类为一类。我们将使用一个问题数据库，其中每个问题都用问题的内容来做标记（标签）。例如，“谁是秦始皇？”将是一个问题，它的标签将是“人”。关于问题句子分类的数据集，国外有些现成的数据集。例如，http://cogcomp.org/Data/QA/QC/ 上提供的句子分类数据集，我们可以找到 1000 个训练句子及其各自的标签和 500 个测试句子。

这里，我们将参考大家所熟知的 Yoon Kim 于 2014 年发表的论文《*Convolutional Neural Networks for Sentence Classification*》，来解读 CNN 在 NLP 任务中的工作价值。然而，使用 CNN 进行句子分类与我们讨论过的 MNIST 示例有些不同，因为现在有些操作（例如，卷积和池化）是发生在一维而不是二维层面上。此外，这里的池化运算也将与正常的池化运算有一定的差异，在后面就会看到。相关完整代码，可查看 ch5 文件夹下的 5_cnn_sentence_classification.ipynb 文件。

5.10.1 CNN 结构部分

现在我们讨论一下用于句子分类的 CNN 的技术细节。首先，我们将讨论如何把数据或句子转换为 CNN 可以轻松处理的首选格式。接下来，我们讨论卷积和池化运算如何适用于句子分类，最后，我们再讨论所有这些组件是如何连接的。

1. 数据转换

假设一个由 p 词所组成的句子。首先，我们用一些特殊的词填充句子（如果句子的长度是小于 n 的），将句子长度设置为 n 个词，其中 $n \geqslant p$。接下来，我们将通过大小为 k 的向量来表示句子中的每个词，其中该向量可以用 One-Hot 编码表示，也可以使用 Skip-Gram、CBOW 或 GloVe 等模型来表示。然后，一组大小为 b 的句子可以用 $b \times n \times k$ 矩阵表示。

下面来看一个例子：

- Bob and Mary are friends.
- Bob plays soccer.
- Mary likes to sing in the choir.

在这个例子中，第三个句子的单词最多，所以让我们设置 $n=7$，这是第三个句子中的单词数。接下来，让我们看一下每个单词的 One-Hot 编码表示情况。由于整体情况下这里有 13 个不同的单词，因此我们得到如下表示：

```
Bob: 1,0,0,0,0,0,0,0,0,0,0,0,0
and: 0,1,0,0,0,0,0,0,0,0,0,0,0
Mary: 0,0,1,0,0,0,0,0,0,0,0,0,0
```

而且，出于同样的原因，$k=13$。通过这种表示，我们可以将三个句子表示为大小为 3×7×13 的三维矩阵，如图 5-37 所示。

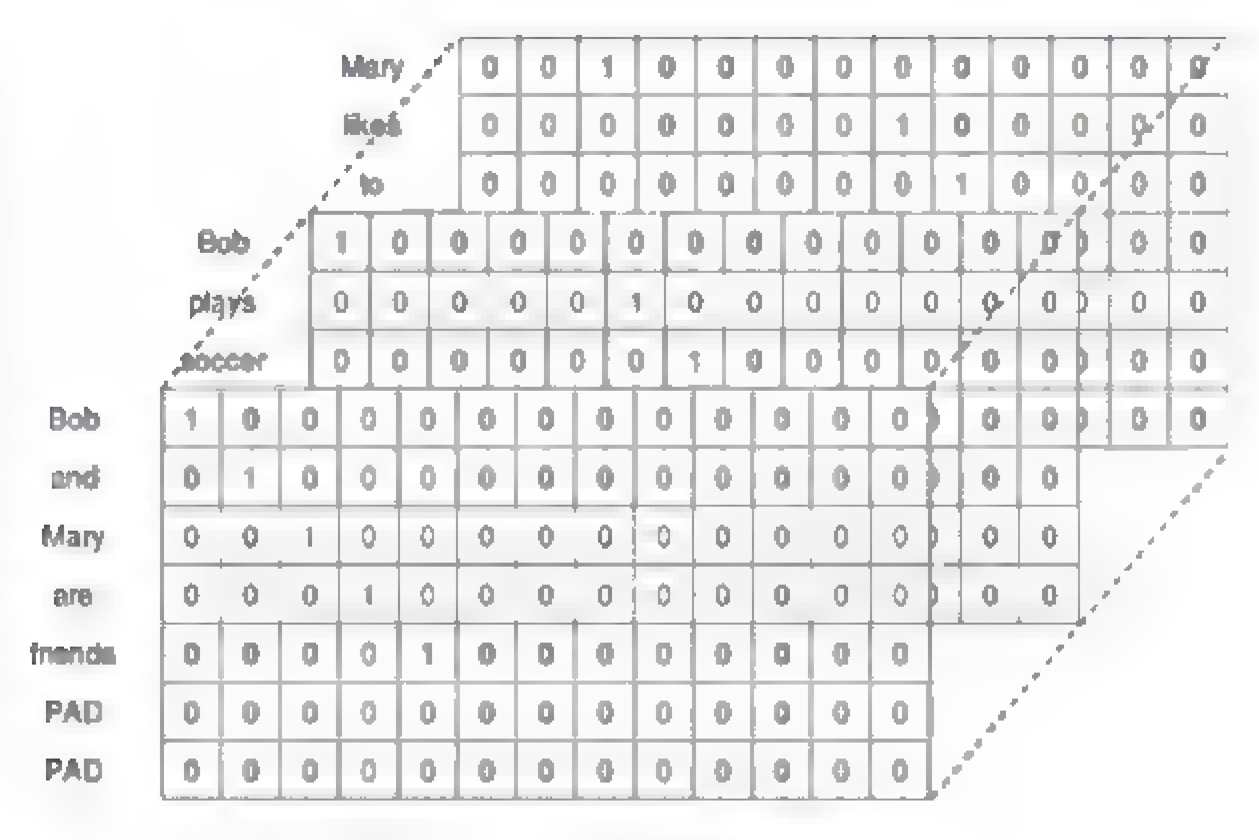


图 5-37　句子矩阵示意图

2. 卷积运算部分

如果我们忽略批量大小，也就是说，如果假设一次只处理一个句子，那么我们的数据将是一个 $n \times k$ 矩阵，其中 n 是填充后每个句子的单词数、k 是一个词向量的维度。在上面的例子中，这个数据是 7×13 矩阵。

现在我们把卷积权重值矩阵定义为 $m \times k$ 的大小，其中 m 是一维卷积运算的过滤器的大小。通过把大小为 $n \times k$ 的输入 x 与大小为 $m \times k$ 的权重值矩阵 W 做卷积运算，我们将产生大小为 $1 \times n$ 的 h 输出，如下所示：

$$h_{(i,1)} = \sum_{j=1}^{m} \sum_{l=1}^{k} w_{(j,l)}\, x_{(i+j-1,l)}$$

这里，$w_{(i,j)}$ 是 W 的第(i, j)个元素，我们将用零填充 x，使得 h 的大小为 $1 \times n$。此外，我们将给出此运算更简单的定义，如下所示：

$$h = W * x + b$$

这里，*定义了卷积运算（带填充），我们将添加一个额外的标量偏差 b。图 5-38 给出了这个运算的说明。

然后，为了学习丰富的特征集合，我们提供了具有不同卷积过滤器大小的并行层。每个卷积层输出一个大小为 $1 \times n$ 的隐藏向量，我们将连接这些输出以形成下一个大小为 $q \times n$ 的层的输入，其中 q 是我们使用的并行层数。q 越大，模型的性能越好。

卷积的值可以用以下方式理解。想一下电影评级学习问题（有两个分类，正面的还是负面的），我们有以下句子：

- I like the movie, not too bad
- I did not like the movie, bad

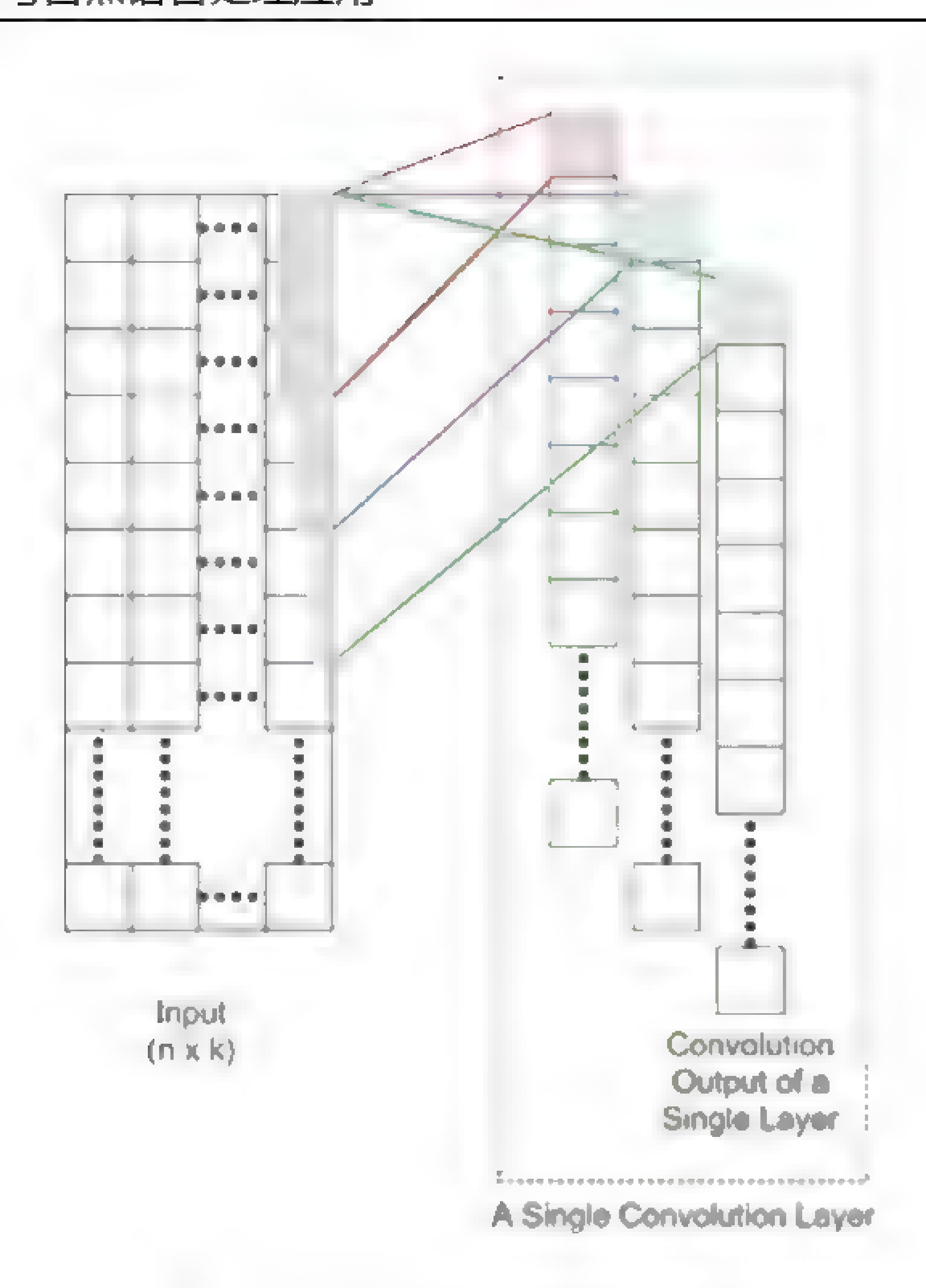


图 5-38　句子分类的卷积运算示意图

现在想象一个大小为 5 的卷积窗口。让我们根据卷积窗口的移动来装箱单词（把这些词存储起来）。

句子“I like the movie, not too bad”的情况如下所示：

```
[I, like, the, movie, ',']
[like, the, movie, ',', not]
[the, movie, ',', not, too]
[movie, ',', not, too, bad]
```

句子“I did not like the movie, bad”的情况如下所示：

```
[I, did, not, like, the]
[did, not ,like, the, movie]
[not, like, the, movie, ',']
[like, the, movie, ',', bad]
```

对于第一句话，如下所示的窗口表示的评级为正面的：

```
[I, like, the, movie, ',']
[movie, ',', not, too, bad]
```

然而，对于第二句话，以下窗口给出的评级是负面的：

```
[did, not, like, the, movie]
```

由于保留了空间性，因此我们能够看到这样的模式有助于对评级进行分类。例如，如果使用诸如词袋（bag-of-words）之类的技术来计算丢失空间信息的句子表示，那么得到的句子表示将是非常相似的。卷积运算在保留句子的空间信息方面起着非常重要的作用。

通过具有不同过滤器大小的 q 个不同层，神经网络学习提取了具有不同大小短语的评级，从而改善了模型的性能。

5.10.2　池化运算

池化运算被设计为对先前讨论的并行卷积层产生的输出进行二次采样，它是通过以下方式来实现的。

假设最后一层 h 的输出大小为 $q \times n$。随着时间的推移，该池化运算将产生大小为 $q \times 1$ 的输出 h'，具体计算如下：

$$h'_{(i,1)} = max(h^{(i)})$$

其中，$1 \leqslant i \leqslant q$。

这里，$h^{(i)} = W^{(i)} * x + b$是由第 i 个卷积层产生的输出，$W^{(i)}$是属于该层的权重值集合。简而言之，随时间的变化，池化运算通过连接每个卷积层的最大元素来创建向量。我们将在图 5-39 中说明此运算。

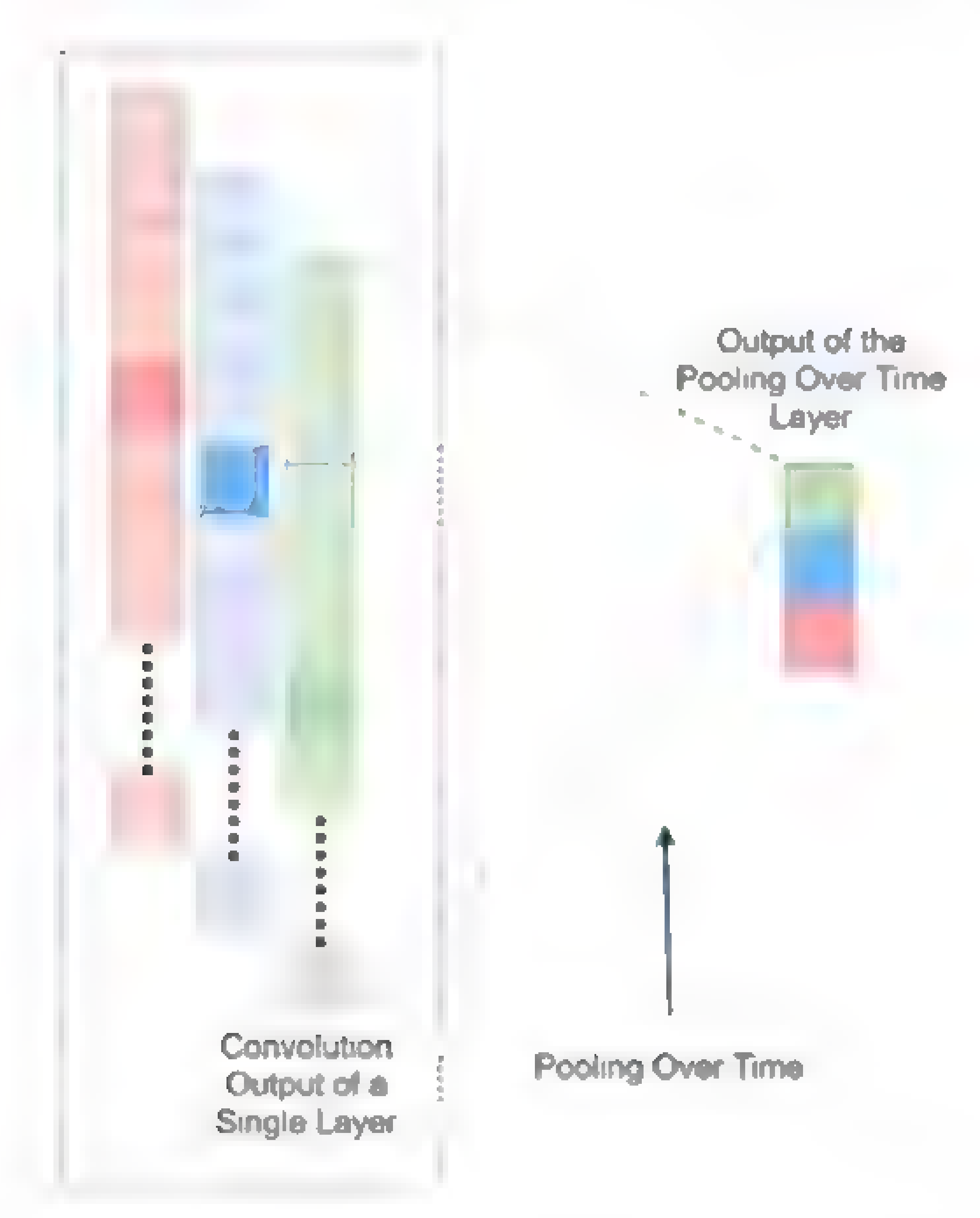


图 5-39　用于句子分类的池化运算随着时间变化的示意图

注意：在图 5-39 中，我们用较暗的颜色来表示并行卷积层的最大值。

最终我们通过组合这些运算得到了图 5-40 所示的架构。

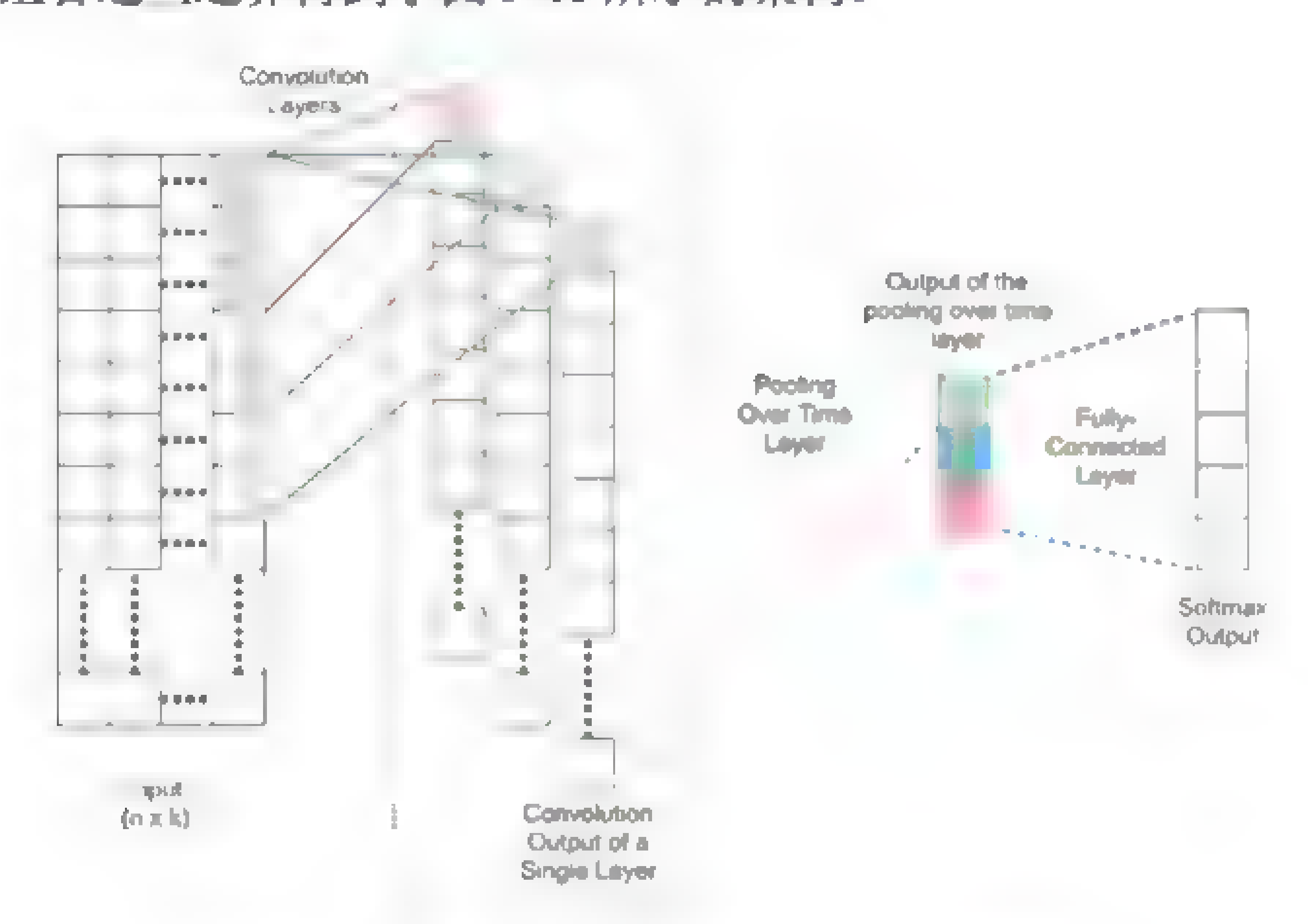


图 5-40 用于句子分类的 CNN 架构示意图

5.10.3 利用 CNN 实现句子分类

首先，我们将定义输入和输出。输入批量句子，其中的词由 One-Hot 编码的词向量表示。我们知道词向量会提供比 One-Hot 编码表示更好的性能，但是，为简单起见，我们这里使用 One-Hot 编码表示：

```
    sent_inputs =
tf.placeholder(shape=[batch_size,sent_length,vocabulary_size],dtype=
                              tf.float32,name='sentence_inputs')
    sent_labels = tf.placeholder(shape=[batch_size,num_classes],
dtype=tf.float32,name='sentence_labels')
```

这里，我们定义了三个不同的一维卷积层，其中三个不同的过滤器的大小分别为 3、5 和 7（在 filter_sizes 中作为列表提供），以及它们各自的偏差：

```
    w1 = tf.Variable(tf.truncated_normal([filter_sizes[0],vocabulary_size,1],
stddev=0.02,dtype=tf.float32),name='weights_1')
    b1 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),
name='bias_1')

    w2 = tf.Variable(tf.truncated_normal([filter_sizes[1],vocabulary_size,1],
stddev=0.02,dtype=tf.float32),name='weights_2')
    b2 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),
name='bias_2')

    w3 = tf.Variable(tf.truncated_normal([filter_sizes[2],vocabulary_size,1],
stddev=0.02,dtype=tf.float32),name='weights_3')
```

```
    b3 = tf.Variable(tf.random_uniform([1],0,0.01,dtype=tf.float32),
name='bias_3')
```

现在我们计算出这三个输出，每个输出属于一个卷积层，正如我们刚刚定义的那样。我们可以使用 TensorFlow 中提供的 tf.nn.conv1d 函数来轻松完成计算。我们使用步幅 1（stride=1）和零填充来确保输出与输入具有相同的大小：

```
    h1_1 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w1,stride=1,padding='SAME') + b1)
    h1_2 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w2,stride=1,padding='SAME') + b2)
    h1_3 = tf.nn.relu(tf.nn.conv1d(sent_inputs,w3,stride=1,padding='SAME') + b3)
```

为了计算随时间变化过程中的最大池化，我们需要编写基本函数来在 TensorFlow 中执行相关运算，因为 TensorFlow 中没有执行此运算的本地函数。而实际上，编写这些函数不麻烦。

首先，我们计算每个卷积层中产生的最大值。这会为每个图层生成一个标量：

```
    h2_1 = tf.reduce_max(h1_1,axis=1)
    h2_2 = tf.reduce_max(h1_2,axis=1)
    h2_3 = tf.reduce_max(h1_3,axis=1)
```

然后，我们连接轴 1（宽度）上产生的输出，以产生大小为 batchsize×q 的输出：

```
    h2 = tf.concat([h2_1,h2_2,h2_3],axis=1)
```

接下来，我们定义完全连接的层，这些层将完全连接到由池化层随时间而产生的输出 batchsize×q。在这种情况下，只有一个完全连接的层，这也将是我们的输出层：

```
    w_fc1 = tf.Variable(tf.truncated_normal([len(filter_sizes),num_classes],
stddev=0.5,dtype=tf.float32),name='weights_fulcon_1')
    b_fc1 = tf.Variable(tf.random_uniform([num_classes],0,0.01,dtype=
                                       tf.float32),name='bias_fulcon_1')
```

此处定义的函数将生成随后用于计算神经网络损失的 logits：

```
    logits = tf.matmul(h2,w_fc1) + b_fc1
```

通过将 softmax 激活应用于 logits，我们将获得相关的预测：

```
    predictions = tf.argmax(tf.nn.softmax(logits),axis=1)
```

此外，我们将定义损失函数，即交叉熵损失：

```
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2
(labels=sent_labels,logits=logits))
```

为了优化网络，我们使用名为 MomentumOptimizer 的 TensorFlow 内置优化器：

```
    optimizer = tf.train.MomentumOptimizer(learning_
rate=0.01,momentum=0.9).minimize(loss)
```

我们可以执行这些先前定义的操作，以便优化 CNN 并评估案例中给出的测试数据，模型在该句子分类任务中给出了接近 90%（500 个测试句子）的测试准确率（在 49 个 Epoch 上，训练数据的平均损失: 0.28，测试数据的平均准确率: 88.333，详见代码运行结果）。

到这里，关于使用 CNN 进行句子分类的讲解就结束了。我们首先讨论了如何将一维卷积运算与被称为 pooling over time 的特殊池化运算相结合，以实现基于 CNN 架构的句子分类器。最后，我们讨论了如何使用 TensorFlow 来实现这样的 CNN，我们看到它在句子分类中表现良好。

了解刚才解决问题的思路，有助于我们在现实世界中把这种解决思路利用起来。假设我们有一份关于罗马历史的大型文档，并且我们想要在不阅读整篇文档的情况下认识 Julius Caesar。在这种情况下，我们刚刚实现的句子分类器便可以作为一个工具来总结只对应于一个人的句子，不必阅读整个文档。

句子分类也可用于许多其他任务，一个常见的用途是将电影评论分类为正面的或负面的，这对于自动计算电影评级很有用。句子分类的另一个重要应用是在医学领域，我们可以从包含大量文本的大型文档中提取出临床有用的句子。

5.11 总结

在本章中，我们首先说明了 CNN 的来龙去脉，并对其五个层级结构（输入层、卷积运算层、激励层、池化层、全连接层）和四个基本运算单元（卷积运算、池化运算、全连接运算和识别运算）有了认识。

其次，我们结合 CNN 网络的四个基本运算单元对于 CNN 的五个层级结构进行逐一解析，对其工作原理进行了详细剖析，并讨论了几个与这些运算符相关的超参数，例如过滤器大小、步幅和填充。为了更好地解释 CNN 中的各个组件，我们尽可能详细地给出了对应的图形化展示。

接着，我们给出了几种常见经典卷积神经网络：AlexNet、VGGNet、Google Inception Net 和 ResNet，并逐一从网络思想、结构、特性亮点等方面做了详尽解读，使我们对近些年来听说过的经典卷积网络的发展路径有了一个完整的认识。

最后，我们为了将上述解析的模型思想真正落到代码层面，给出了两个案例：利用 CNN 对 MNIST 数据集进行图片分类和利用 CNN 对句子进行分类。在对 MNIST 数据集进行图片分类中，我们还进行了一些分析，以了解为什么 CNN 无法正确识别某些图像。而在利用 CNN 对句子进行分类中，我们讨论了可用于对句子进行分类的 CNN 架构，并在实际的句子分类任务上进行了测试。

在下一章中，我们将继续讨论用于许多 NLP 任务最流行的神经网络之一：循环神经网络（RNN）。

第 6 章

循环神经网络

在本章中，我们将介绍循环神经网络（Recurrent Neural Network，RNN），它是一类旨在处理输入序列数据的神经网络，专门用于处理序列$x^1, x^2, x^3, \dots, x^T$这样的神经网络。这些输入可以是文本、语音、时间序列，序列中元素的出现取决于其之前出现元素的任何其他内容。RNN 非常灵活，就像卷积神经网络能够很轻松地扩展到具有很大宽度和高度的图像且可以处理大小可变的图像，RNN 能够扩展到更长的序列且多数能够处理可变长度的序列。正因为如此，RNN 已被广泛用于解决诸如语音识别、文本生成、情感分析、图像字幕提取和机器翻译等问题。

其实，RNN 在维护着一个状态变量，以此获取序列数据中存在的各种模式，进而能够对序列数据建模。传统的前馈神经网络不具有这种能力，除非用获取序列中存在的重要模式的特征表示来对数据进行表示。然而，对这样的特征表示是非常困难的。前馈网络对序列数据进行建模的另一种方案是让时间/顺序中的每个位置具有单独的参数集，但是这样一来将大大增加内存的压力。所以，这里我们就需要利用到二十世纪八十年代机器学习和统计学思想的优点：在模型的不同部分共享参数。因为参数共享使模型能够扩展到不同形式的样本且可以泛化。反之，在每个时间点上均有单独的参数，这不仅难以做到泛化训练时未曾出现过的序列长度，而且不能在时间上共享不同序列长度以及各个位置的统计强度。比如，这里有一个固定长度的句子，我们要去训练它，传统的全连接前馈网络会给每个输入特征单独分配一个参数，这样对于句子中每个位置的所有语音规则均需要学习，而 RNN 由于在多个时间步长内能够共享相同的权重值，因此不必对句子中每个位置的语言规则都学习一遍。

在本章中，我们将深入探讨 RNN 的细节。首先，我们将通过计算图及其循环图的展开的计算，引出循环网络。在此之后，我们将通过序列数据模型引出 RNN 结构的简要计算，并从数学层面给出详解。我们还将深入研究相关的基础方程，例如 RNN 的输出计算和参数更新规则，并讨论 RNN 应用的几种变体：一对一、一对多、多对一和多对多的 RNN。最后将通过一个案例具体介绍使用 RNN 基于训练数据集来生成新文本，并讨论 RNN 的一些局限性。在计算和评估生成的文本之后，我们将讨论 RNN 的扩展，称为 RNN-CF，与常规 RNN 相比记忆更长，最后通过 RNN-CF 给出案例的优化，并对它进行解读和分析。

6.1 计算图及其展开

计算图是一种用于表达和评估数学表达式的方式，被定义为有向图，涉及的节点对应于数学运算。例如，将输入、参数映射到输出和损失的计算等。这里我们结合 RNN 计算得到的重复结构进行阐释，这些重复结构一般对应于一个事件链。下面给出动态系统框架下的经典形式：

$$s_t = f(s_{(t-1)}; \theta) \tag{6.1}$$

这里s_t代表系统状态。

显然式（6.1）是循环的，对于有限的 t 而言，式（6.1）逐步展开最终会得到不涉及循环的表达形式。因此，这里我们能够使用传统的有向无环计算图给出相关表达形式，如图 6-1 所示。

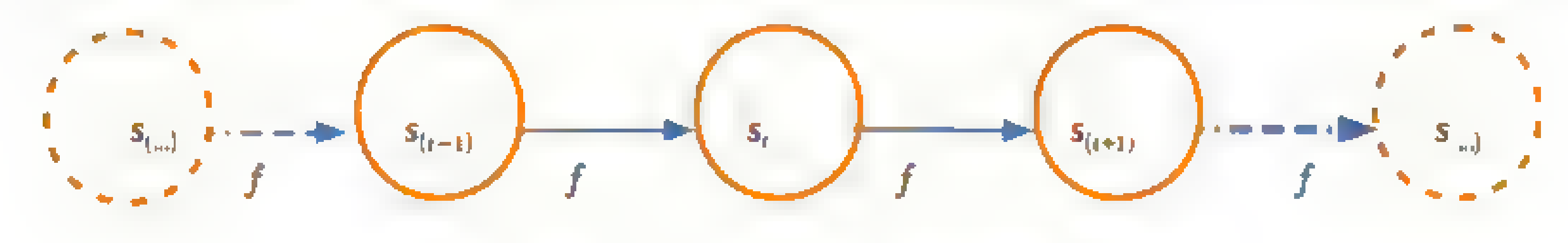


图 6-1　式（6.1）的计算图

图 6-1 将式（6.1）代表的经典动态系统用展开的计算图表示出来。每个节点表示在某个时刻 t 的状态，并且函数 f 将 t 处的状态映射到 $t+1$ 处的状态。所有时间步长都使用相同的参数（用于参数化 f 的相同 θ 值）。

而在有外部信号进入的情况下，由外部信号x_t驱动的动态系统表达式如下：

$$s_t = f(s_{(t-1)}, x_t; \theta) \tag{6.2}$$

显然当前状态涵盖了整个过去的序列信息。

实际上，涉及循环的任何函数本质上可以被认为是一种循环神经网络，且多数循环神经网络可以由式（6.2）来定义隐藏单元的值。下面给出状态变量 h 的公式，也是多数 RNN 中的典型公式，如式（6.3）所示：

$$h_t = f(h_{(t-1)}, x_t; \theta) \tag{6.3}$$

结合式（6.3），我们给出 RNN 在增加外部信号 x 时读取状态信息 h 并进行预测的示意图，如图 6-2 所示。

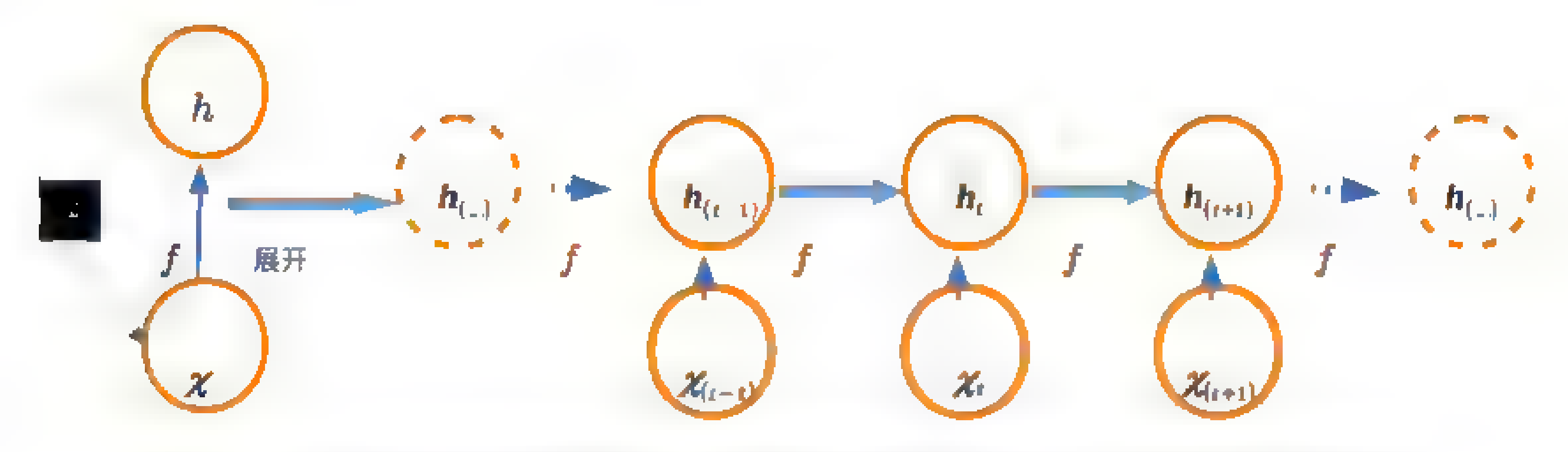


图 6-2　RNN 在增加外部信号 x 时读取状态信息 h 并进行预测的示意图

图 6-2 中没有输出，此循环网络仅处理来自输入 x 的信息，将其合并到经过时间向前传播的状态 h。左图为回路原理图，黑色方块表示单个时间步长的延迟。右图为同一网络被展开的计算图，其中每个节点现在与一个特定的时间实例相关联。

这里，图 6-2 左图回路图中黑色方块代表在时刻 t 状态到时刻 $t+1$ 状态的单个时刻延迟中的相互作用。

我们发现，对于式（6.3）的描述可以有两种方式，一种是左图代表回路原理的循环图，另一种是右图代表的展开计算图。对于左图，网络给出了实时运算回路，当前状态会影响其未来状态；对于右图展开的计算图而言，里面的任意一个组件均有多个不同的变量表达式，一个时间步长一个变量，表示该时间点上组件的状态情况，每个变量为计算图中的一个独立节点。下面我们给出一个函数g_t来代表经 t 步展开后的循环：

$$h_t = g_t(\chi_t, \chi_{(t-1)}, \chi_{(t-2)}, \ldots, \chi_2, \chi_1) \tag{6.4}$$

$$=f(h_{(t-1)}, \chi_t; \theta) \text{ ((} \tag{6.5}$$

结合式（6.4）和式（6.5），对于函数g_t而言，它能够将所有之前的序列$(\chi_t, \chi_{(t-1)}, \ldots, \chi_2, \chi_1)$作为输入以生成当前状态。同时我们可以看到，展开后的循环结构允许我们把函数g_t分解为重复应用的函数f。显然，展开过程会带来以下好处：

（1）无论序列长度如何，训练好的模型一直具有同样的输入大小，这是由于它指定的不是在可变长度的历史状态上运算，而是由一种状态到另一种状态的转移。

（2）在每个时间步长我们可以使用相同参数的相同转移函数f。

综上所述，循环图和展开图都有各自的用途。循环图很简洁，而展开图可以清晰地描述其中的计算流程，还可以借助显式信息流的流动路径解释信息在时间上向前（计算输出和损失）和向后（计算梯度）的作用。

6.2 RNN 解读

6.2.1 序列数据模型

在现实世界中，我们能够接触到很多像$\{\chi_1, \chi_2, \chi_3, \ldots, \chi_T\}$、$\{y_1, y_2, y_3, \ldots, y_T\}$这样的序列式数据，并且这些序列式数据在多个领域均有相应的应用，比如：

（1）在自然语言处理中，χ_1可以作为第一个词，χ_2作为第二个词，后面以此类推。

（2）在语音处理中，$\chi_1, \chi_2, \chi_3 \ldots$是每帧的声音信号。

（3）在时间序列问题中，像每天的股票价格、每天的气温等。

结合 6.1 节的内容，我们可以给出χ和y的函数关系，如下：

$$h_t = f_1(h_{(t-1)}, \chi_t; \theta) \tag{6.6}$$

$$y_t = f_2(h_t; \varphi) \tag{6.7}$$

我们可以将f_1、f_2视为生成 x 和 y 的真实模型的近似值，更细一点讲，将式（6.6）、（6.7）扩展如下：

$$y_t = f_2(f_1(\chi_t, h_{(t-1)}; \theta); \varphi) \quad (6.8)$$

假设这里 t=4，由式(6.8)可知y_4为：

$$y_4 = f_2(f_1(\chi_4, h_3; \theta); \varphi) \quad (6.9)$$

下面对式(6.9)做进一步展开，我们可以得到最终结果（为了表达式清晰起见，这里省略θ和φ）：

$$y_4 = f_2(f_1(\chi_4, f_2(f_1(\chi_3, f_2(f_1(\chi_2, f_2(f_1(\chi_1, h_0)))))))) \quad (6.10)$$

我们将式(6.10)在图中进行展示，如图 6-3 所示。

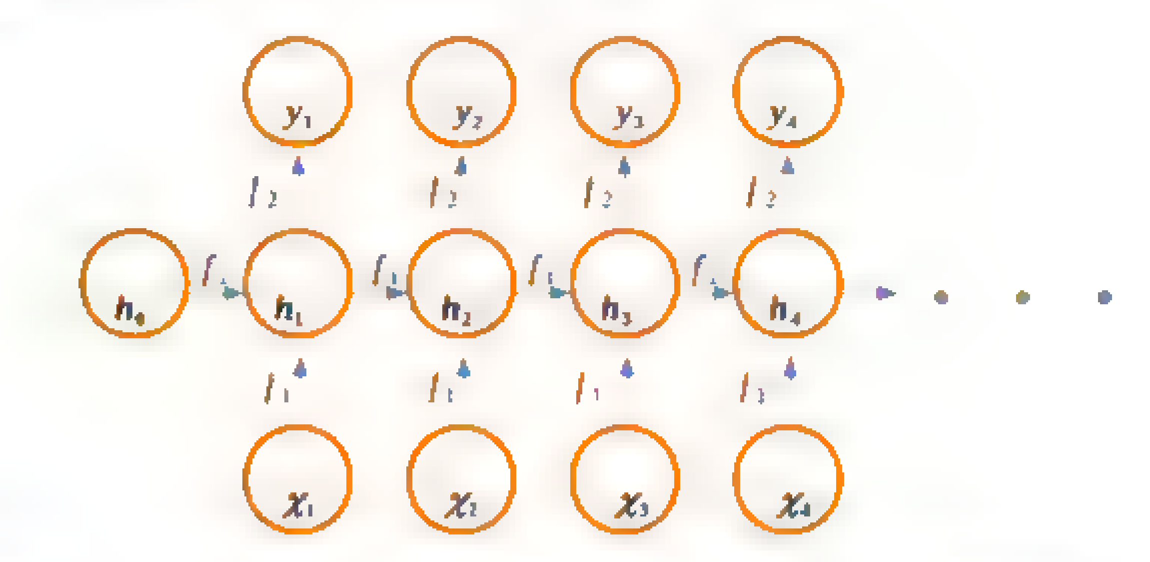


图 6-3　χ_t和y_t之间的关系图，随着 t 的增加，后面继续扩展

这里，一个箭头表示对相应向量做一次类似$f(W\chi+b)$的变换。

对于任何给定的时间步长 t，我们一般以图 6-4 来概括。

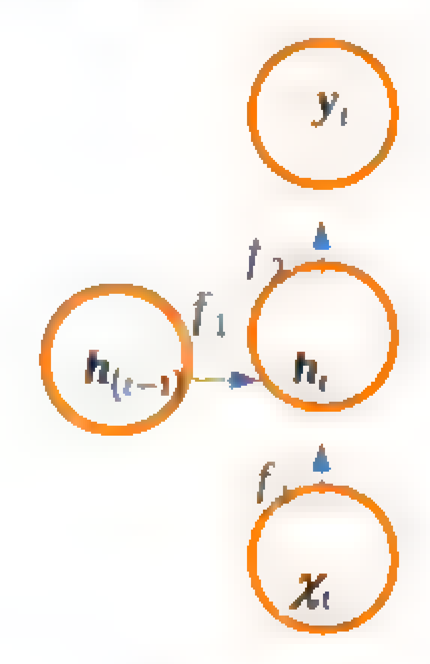


图 6-4　一个 RNN 结构的单步计算示意图

尽管如此，对于h_{t-1}而言，实际上是接收χ_t之前的h_t。也就是说，h_{t-1}实际上是一个时间步长之前的h_t。所以，我们可以使用循环连接来表示h_t的运算情况，如图 6-5 所示。

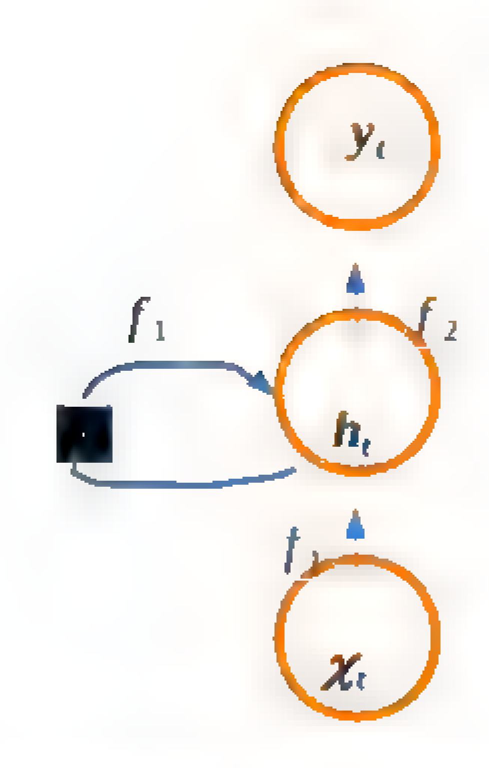


图 6-5　利用循环连接对 RNN 结构的单步计算示意图

注：这里的黑色框与 6.1 节中的含义一样，即代表单个时间步长的延迟

在图 6-5 中，我们可以看到，这里概括了序列$\{\chi_1,\chi_2,\chi_3,\dots,\chi_T\}$映射到序列$\{y_1,y_2,y_3,\dots,y_T\}$的方程链。更进一步讲，通过$h_{(t-1)}$、$h_t$、$\chi_t$可以得到任意的$y_t$，这也是 RNN 的核心思想所在。

6.2.2　数学层面简要解读 RNN

现在让我们继续研究一下 RNN 到底是什么，并为 RNN 中的计算定义数学方程式。让我们从派生的两个函数开始，作为从 x_t 学习 y_t 的函数逼近器：

$$h_t = f_1(x_t, h_{(t-1)}; \theta)$$
$$y_t = f_2(h_t; \varphi)$$

正如我们所看到的那样，神经网络由一组权重值、偏差和非线性激活函数所组成，我们可以改写前面的关系如下：

$$h_t = \tanh(U\chi_t + Wh_{(t-1)})$$

这里，$tanh()$是 tanh 激活函数，U 是大小为 m×d 的权重值矩阵，其中 m 是隐藏单元的数量，d 是输入的维数。此外，W 是大小为 m×m 的权重值矩阵，它创建了从 $h_{(t-1)}$ 到 h_t 的循环连接。上面的y_t关系式可由以下等式给出：

$$y_t = softmax(Vh_t)$$

这里，V 是大小为 c×m 的权重值矩阵，c 是输出的维数（可以是输出类的数量）。这样一来，我们就可以给出这些权重值形成 RNN 的路径图，如图 6-6 所示。

综上所述，我们已经阐释了通过计算节点图来表示一个 RNN，并大致学习了 RNN 背后的数学逻辑关系。下一步，我们将对 RNN 的权重值进行优化（或者训练），以便更好地学习序列数据样本。

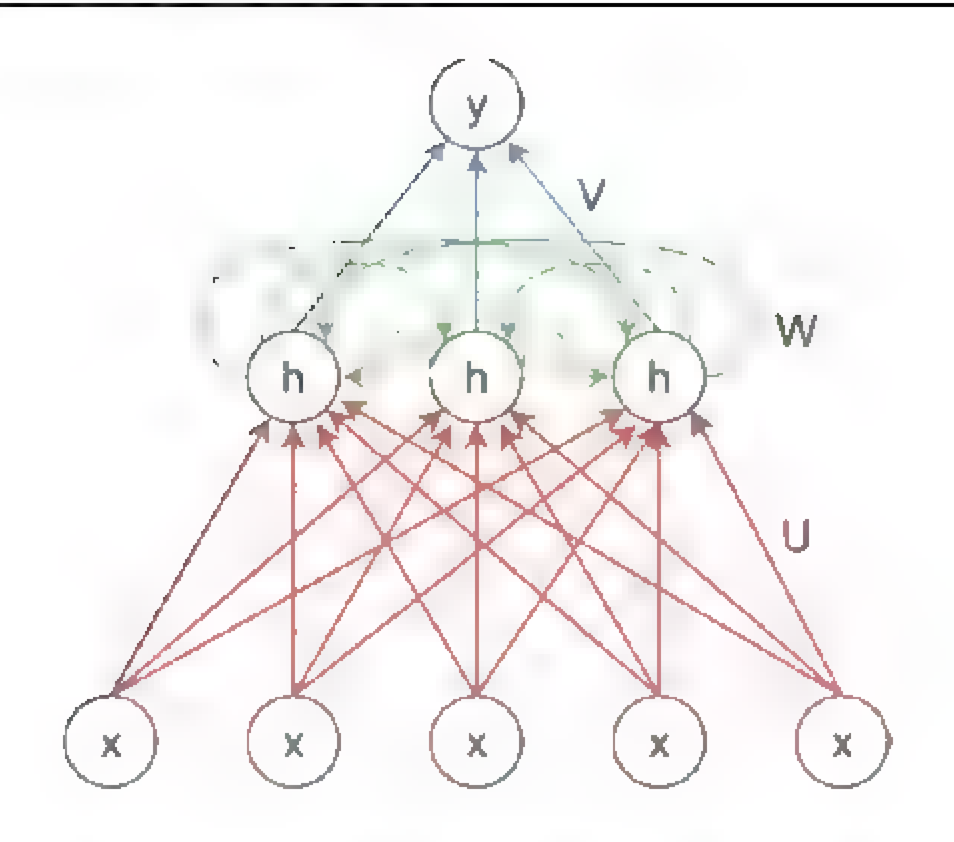


图 6-6 RNN 的结构示意图，边表示计算

6.3 通过时间的反向传播算法

对于训练 RNN，使用一种特殊形式的反向传播，称为通过时间的反向传播（Back Propagation Through Time，BPTT）。本节首先介绍反向传播（BP）的工作原理；然后讨论为什么反向传播不能直接应用于 RNN，以及反向传播如何适应 RNN 后可以产生 BPTT；最后讨论 BPTT 中存在的两个主要问题。

6.3.1 反向传播工作原理

反向传播是用于训练前馈神经网络的技术，在反向传播中，执行以下操作：

（1）计算给定输入的预测。

（2）通过将预测与对应实际值进行比较来计算预测误差 E（例如均方误差和交叉熵损失）。

（3）在所有层上的所有权重值$w_{(i,j)}$（第 i 层上第 j 个权重值）的梯度$\frac{\partial E}{\partial w_{(i,j)}}$的反方向调整一个小的步长，更新前馈网络的权重值，从而使得步骤（2）中的计算损失最小化。

为了方便理解，我们这里给出前馈网络的示意图且省去了输入和输出的时间部分。这里有两个单一权重值：U 和 V，且计算出两个输出 h 和 y，具体如图 6-7 所示（这里假设模型中不存在非线性）。

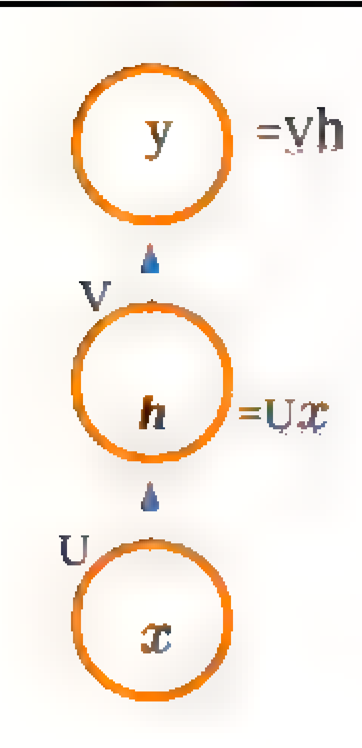


图 6-7　前馈网络计算

所以，我们可以利用链式法则计算$\frac{\partial E}{\partial U}$，如下：

$$\frac{\partial E}{\partial U}=\frac{\partial (y-l)^2}{\partial y}\frac{\partial (vh)}{\partial h}\frac{\partial (Ux)}{\partial U}$$

进一步简化得到：

$$\frac{\partial E}{\partial U}=\frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial U}$$

这里，l 是数据点 x 的正确标签。此外，我们假设均方误差为损失函数。这里的所有内容都已定义，计算$\frac{\partial E}{\partial U}$就较为简单了。

6.3.2　直接使用反向传播的局限性

现在，我们对图 6-8 中的 RNN 进行类似的操作，这里新增加了一个循环权重值 W。

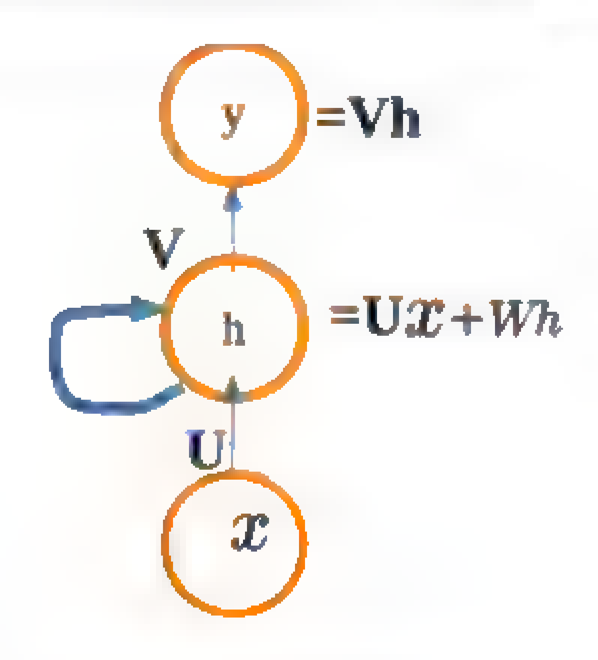


图 6-8　RNN 的计算

下面对$\frac{\partial E}{\partial W}$的计算使用链式法则，得到：

$$\begin{aligned}\frac{\partial E}{\partial W}&=\frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial W}\\&=\frac{\partial (y-l)^2}{\partial y}\frac{\partial (vh)}{\partial h}\left(\frac{\partial (Ux)}{\partial W}+\frac{\partial (Wh)}{\partial W}\right)\end{aligned}$$

这里，由于$\frac{\partial (Wh)}{\partial W}$是一个递归项，本身就会带来一些问题。最终会得到无穷多的导数项，因为 h 是递归的（也就是说，计算 h 包括 h 本身），且 h 不是常数并依赖于 W。在这种情况下，我们需要利用基于时间展开输入序列 x 来解决，为每个输入x_t创建 RNN 的副本并分别计算每个副本的导

数，通过对梯度求和将它们回滚以更新权重值。接下来简要讨论一下具体情况。

6.3.3 通过反向传播训练 RNN

计算 RNN 反向传播的技巧是不考虑单个输入，而是考虑完整的输入序列。如果我们在时间步长为 4 上时计算$\frac{\partial E}{\partial w}$，将得到以下结果：

$$\frac{\partial E}{\partial w}=\sum_{j=1}^{3}\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_j}\frac{\partial h_j}{\partial w}$$

这意味着我们需要一直计算到第 4 个时间步长的所有时间步长的梯度之和。换句话说，我们将首先展开序列，以便可以计算每个时间步长 j 的$\frac{\partial h_4}{\partial h_j}$和$\frac{\partial h_j}{\partial w}$，这是通过创建 RNN 的 4 个副本来完成的。因此，要计算$\frac{\partial h_t}{\partial h_j}$，就需要(t-j+1)份的 RNN。接着将这些副本卷起到单个 RNN，通过之前所有时间步长获得的梯度归总求和，并用梯度$\frac{\partial E}{\partial w}$更新 RNN。但是，这里有一个问题，随着时间步长数的增加，相应计算成本也会增加。为了解决这个问题，我们将会使用截断 BPTT（TBPTT）来优化模型，这是 BPTT 的近似值。

6.3.4 截断 BPTT

在截断 BPTT 中，我们只计算固定数量的 T 个时间步长的梯度，更具体地说，在计算$\frac{\partial E}{\partial w}$时，对于时间步长 t，我们只计算导数，直到 t-T：

$$\frac{\partial E}{\partial w}=\sum_{j=t-T}^{t-1}\frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_j}\frac{\partial h_j}{\partial w}$$

这比标准 BPTT 计算效率高得多。在标准 BPTT 中，对于每个时间步长 t，我们计算直到序列最开始的导数，但随着序列长度变得越来越长（例如逐字处理文本文档），这种计算是难以实现的。而在截断的 BPTT 中，我们仅向后计算固定数量的导数，可以想象，随着序列变大，计算成本不会改变。

6.3.5 BPTT 的局限性——梯度消失和梯度爆炸

我们有办法计算循环权重值的梯度并给出有效的近似值，如 TBPTT，这让我们对于平稳地训练 RNN 感到担忧，这里的计算其实出现了其他问题。

为了了解其中的原因，让我们在$\frac{\partial E}{\partial w}$中进行相关扩展，如下所示：

$$\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial w}=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial(Ux+Wh_3)}{\partial h_1}\frac{\partial(Ux+Wh_0)}{\partial w}$$

由于我们知道反向传播的问题来自循环连接层，让我们忽略 Ux 项，上式后边变为：

$$\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial w}=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial(Wh_3)}{\partial h_1}\frac{\partial(Wh_0)}{\partial w}$$

下面我们对h_3继续展开简单运算，得到：

$$\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W}=\frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}h_0W^3$$

这里我们看到只有 4 个时间步长便产生一个W^3，因此在时间步长为“第几个”时，它将变为$W^{(n-1)}$。下面从两个方向对 W 进行极值分析，以此探讨梯度消失和梯度爆炸的成因。

1. 梯度消失

假设我们在 n＝100 的时间步长上将 W 初始化得非常小（比如 0.00001），那么梯度将会无穷小（比如 0.1^{500}）。另外，由于计算机在表示数值方面的精度有限，因此将此更新（数值下溢）忽略，这被称为消失的梯度。解决消失的梯度问题并不简单。这里没有简单的方法来重塑梯度，以便它能够在时间上正确传播。在某种程度上，解决梯度消失问题的方法是谨慎使用权重值初始化（例如 Xavier 初始化），或使用基于动量的优化方法（也就是说，除了当前的梯度更新外，我们还添加了一个额外项，即所有过去梯度的累积，称为速度项（Velocity Term））。然而，正如我们将在第 7 章“长短期记忆网络”中看到的那样，已经引入了更多有规则的解决方法，例如对标准 RNN 的不同结构调整等。

2. 梯度爆炸

假设我们将 W 的取值初始化得非常大（比如 1000.00），则在时间步长 n 取 100 时，梯度将变得非常大（比例为10^{300}）。这会导致数值非常不稳定，在 Python 中将会获得诸如 Inf（无穷大）或 NaN（不是一个数值）之类的结果，这就称为梯度爆炸。

其实，问题损失表面产生的复杂性也可能引起梯度爆炸。由于输入的维数以及模型中存在的大量参数（权重值），复杂的非凸损失表面在深度神经网络中非常普遍。图 6-9 显示了 RNN 的损失表面，并突出显示了具有很高曲率表面的存在。如果优化方法与这样的表面接触，梯度就会爆炸，如图 6-9 中的实线所示。这可能导致非常差的损失最小化或数值不稳定性或两者兼而有之。在这种情况下，避免梯度爆炸的简单解决方案是在梯度大于某个阈值时将梯度取值调整为足够小。图 6-9 中的虚线表示当我们将梯度调整为某个较小值时会发生什么。

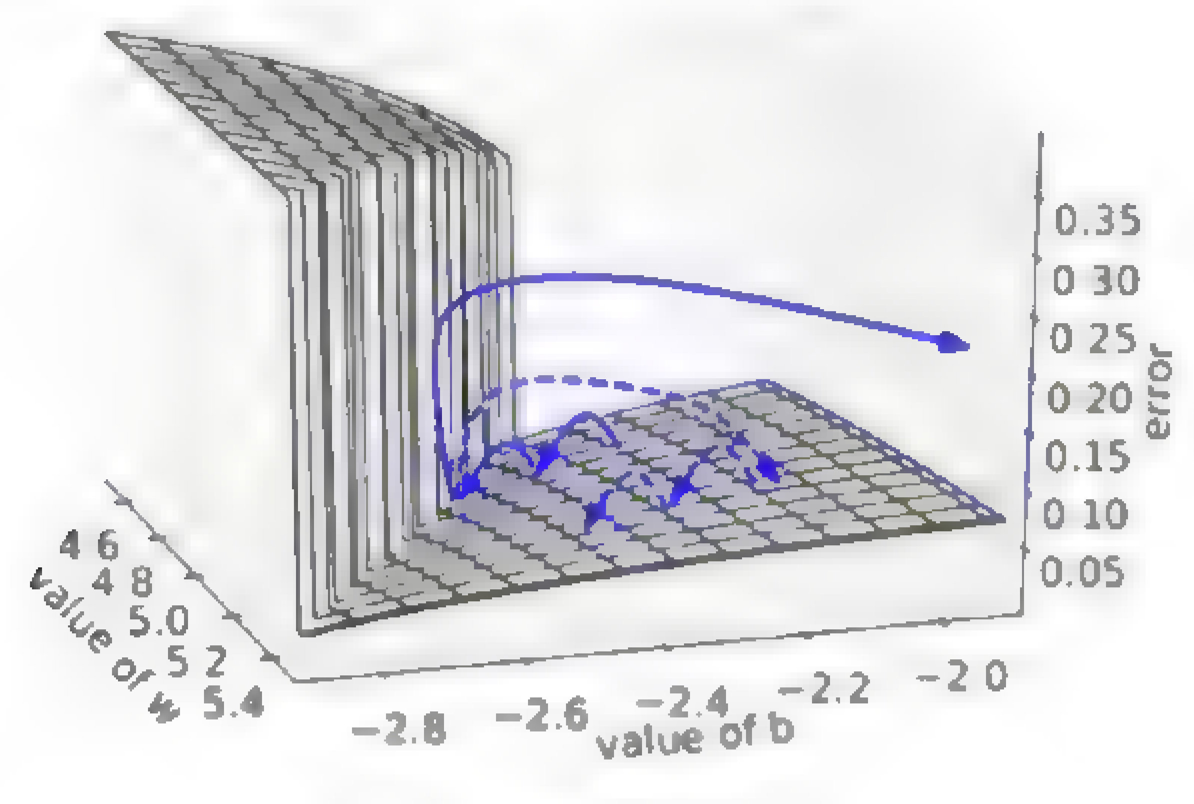


图 6-9　梯度爆炸情况

关于训练 RNN 中的梯度问题，论文《*On the Difficulty of Training Recurrent Neural Networks*》（Pascanu,Mikolov, and Bengio, International Conference on Machine Learning (2013): 1310-1318）做了相关分析，感兴趣的读者可以阅读该论文。

到目前为止，我们讨论的都是一对一映射的 RNN。接下来，我们将对各种映射类型的 RNN 做相关介绍，这些不同映射类型的 RNN 在句子分类、图像字幕提取及机器翻译中有很好的应用。

6.4 RNN 的应用类型

一对一映射的 RNN 中，当前输出取决于当前输入以及先前观察到的输入历史。这意味着存在先前观察到的输入序列和当前输入的输出。然而，在实际的应用中，可能存在这样的情况：输入序列只有一个输出、单个输入产生的输出序列以及序列大小不同的输入序列产生的输出序列。在本节中，我们将对这些情况做相关介绍。

6.4.1 一对一的 RNN

在一对一的 RNN 中，目前的输入取决于先前观察到的输入，如图 6-10 所示。这种 RNN 适用于每个输入都有输出的问题，但输出取决于当前输入和导致当前输入的输入历史情况。这类任务的一个典型例子是股票市场预测。另一个例子是场景分类，其中图像中的每个像素被标记（例如汽车、道路和人的标签）。对于某些问题，有时$x_{(t+1)}$可能与y_t相同。例如，在文本生成问题中，先前预测的词变为预测下一个词的输入。

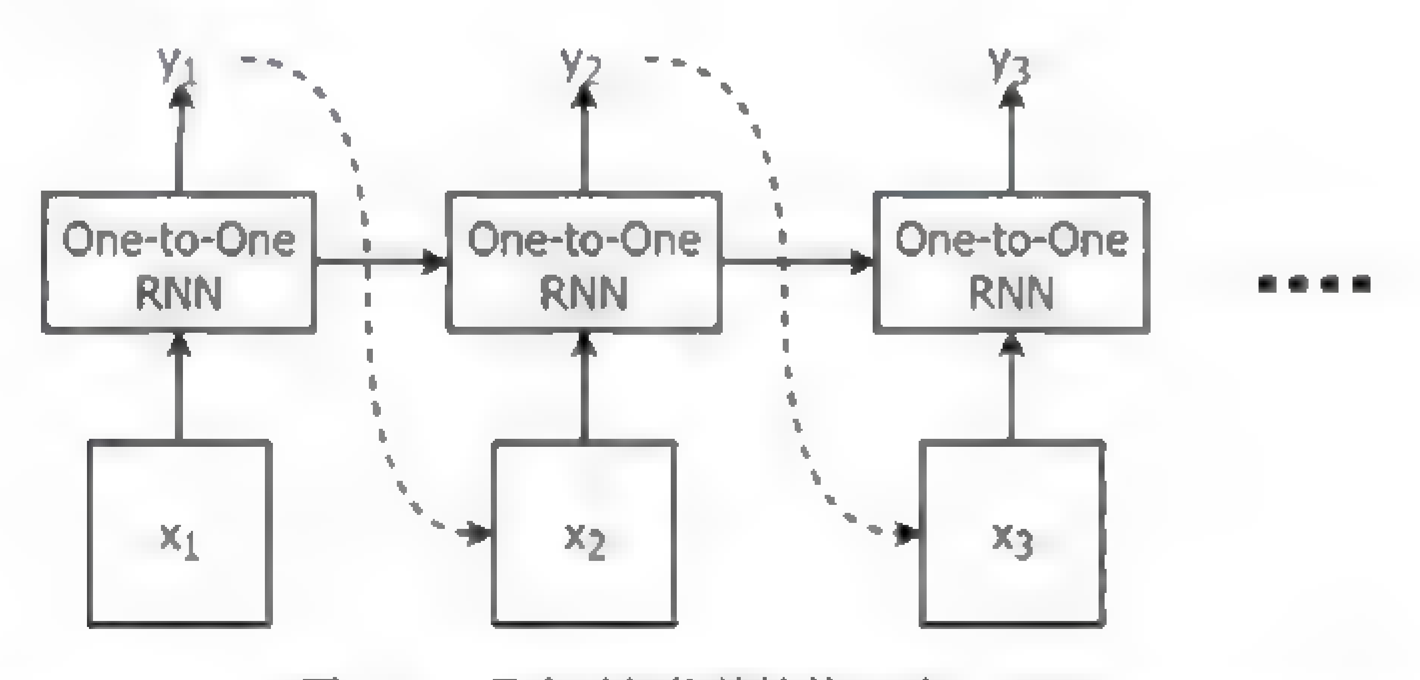


图 6-10 具有时间依赖性的一对一 RNN

注意

虚线表示$x_{(t+1)}$可以与y_t相同，或者$x_{(t+1)}$可以是单独的输入。

6.4.2 一对多的 RNN

一对多的 RNN 采用单个输入并输出序列，如图 6-11 所示。这里假设输入彼此之间是独立的。也就是说，我们不需要有关先前输入的信息来预测当前输入。循环连接还是需要的，因为尽管我们

处理单个输入，但输出依赖于先前输出值的一系列值。这类 RNN 常应用于图像字幕任务，例如对于给定的输入图像，文本标题可以由 5 个或 10 个词所组成。换句话说，RNN 将保持预测词，直到它输出的短语对于描述图像是有意义的。

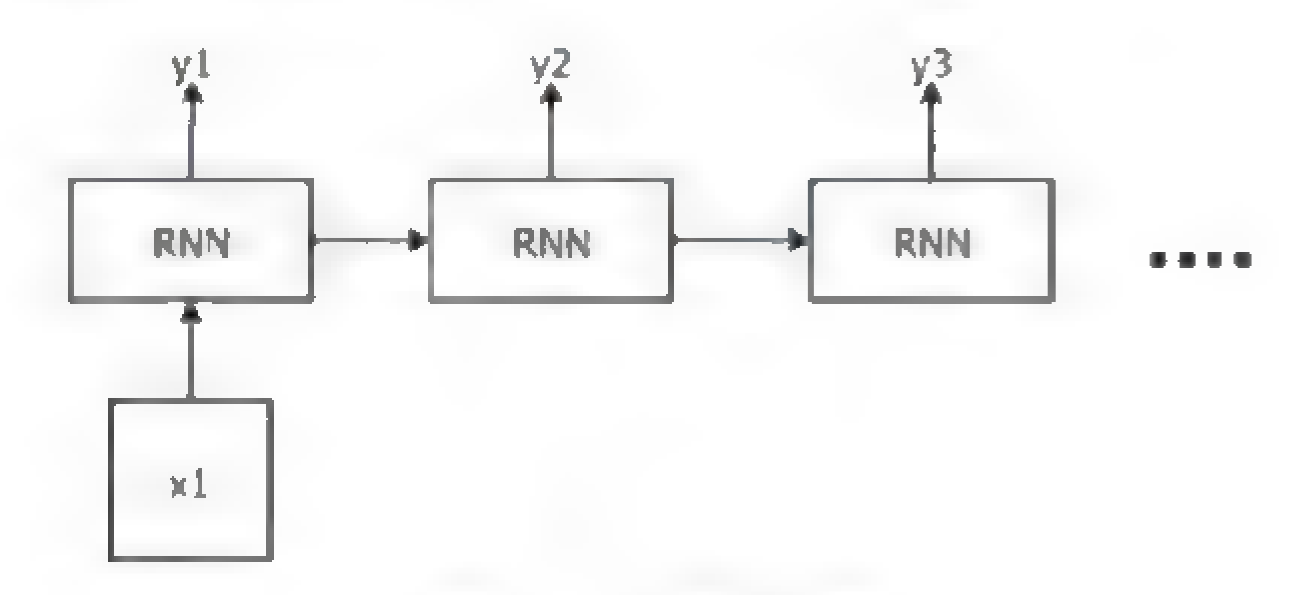


图 6-11　一对多的 RNN

6.4.3　多对一的 RNN

多对一的 RNN 采用任意长度的输入作为输入，并为输入序列产生单个输出，如图 6-12 所示。句子分类就是这样一项任务，可以从多对一的 RNN 中受益。句子是任意长度的词序列，其被视为网络的输入，用于产生将句子分类为一组预定义类之一的输出。句子分类的一些具体例子如下：

（1）将电影评论分类为正面或负面陈述（情感分析）。

（2）根据句子描述的内容（例如人物、对象和位置）对句子进行分类。

多对一的 RNN 的另一个应用是通过一次只处理一块图像并在整个图像上移动窗口来对大尺寸图像进行分类。

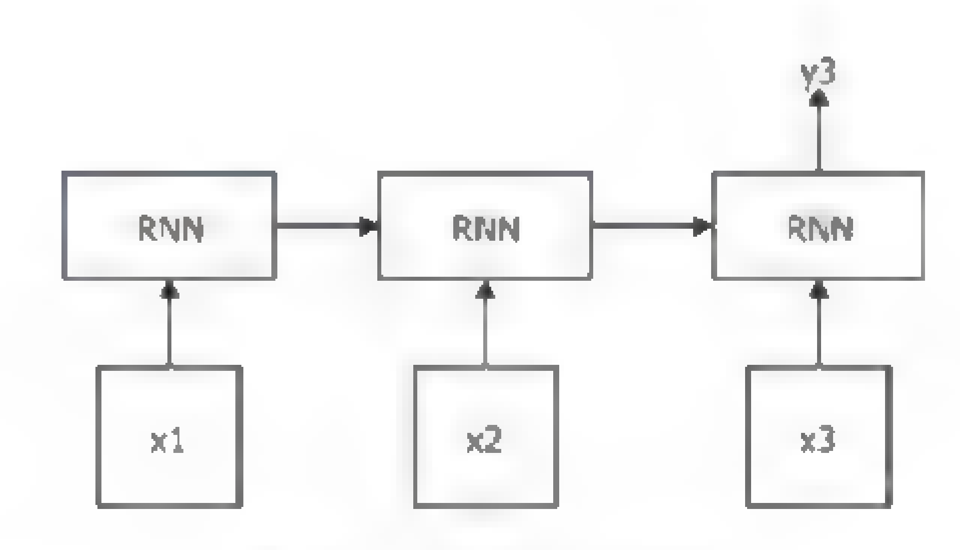


图 6-12　多对一的 RNN

6.4.4　多对多的 RNN

多对多的 RNN 通常从任意长度的输入中产生任意长度的输出，如图 6-13 所示。换句话说，输入和输出不必具有相同的长度。这在机器翻译中特别有用，我们将句子从一种语言翻译成另一种语言时，可以想象，某种语言中的一个句子并不总是与另一种语言的句子对齐。另一个应用是聊天机器人，聊天机器人读取一系列词（用户请求）并输出一系列词（答案）。

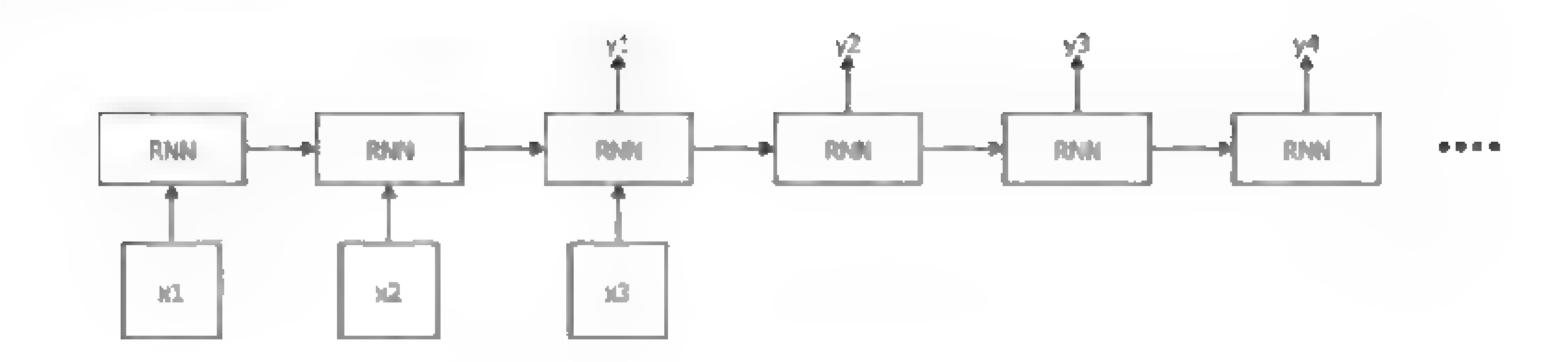


图 6-13 多对多的 RNN

6.5 利用 RNN 生成文本

现在让我们看一下使用 RNN 的第一个例子。在本例中，我们将使用 RNN 生成一个童话故事。这是一对一的 RNN 问题。我们将在一系列童话故事中训练单层 RNN，并要求 RNN 生成一个新故事。对于这项任务，我们将使用一个包含 20 个不同故事的小型文本语料库。这个例子将突显出 RNN 的一个重大局限性：缺乏持久的长期记忆。本例代码详见 ch6 文件夹中的 6_rnn_language_bigram.ipynb 文件。

6.5.1 定义超参数

定义 RNN 所需的几个超参数具体如下：

（1）单个时间步长一次性执行的展开次数。这是输入展开的步数，如截断 BPTT 方法（TBPTT 中的 T（Truncated）是有效训练 RNN 部分）中所讨论的。该数字越大，RNN 的记忆越长。但是，由于梯度消失因素，对于非常高的 num_unroll 值（例如大于 50），此值的效果会消失。注意，增加 num_unroll 也会增加程序的内存需求。

（2）训练数据、验证数据和测试数据的批量大小。较高的批量大小通常会带来更好的结果，因为我们会在每个优化步骤中看到更多数据，然而和 num_unroll 一样，这会导致更高的内存需求。

（3）输入、输出和隐藏层的维度。增加隐藏层的维度通常会带来更好的性能。但请注意，增加隐藏层的大小会导致所有三个权重值集合（即 U、W 和 V）的增加，从而导致占用较高的计算空间。

首先定义展开、批量并测试批量大小：

```
num_unroll = 50
batch_size = 64
test_batch_size = 1
```

接下来定义隐藏层中的单元数（这里使用单个隐藏层 RNN），给出输入和输出大小：

```
hidden = 64
in_size,out_size = vocabulary_size,vocabulary_size
```

6.5.2　随着时间的推移展开截断 BPTT 的输入

正如我们之前看到的，随着时间的推移展开输入是 RNN 优化过程（TBPTT）中的重要部分。因此，下一步是：定义输入如何随时间展开。

这里给出一个例子，假设有一个句子如下：

```
Bob and Mary went to buy some flowers.
```

假设我们以字符的粒度级别处理数据。另外，考虑一批数据，并且展开的步数（num_unroll）是 5。

首先，我们将句子拆分成字符：

```
'B', 'o', 'b', ' ', 'a', 'n', 'd', ' ', 'M', 'a', 'r', 'y', ' ', 'w', 'e', 'n',
't', ' ', 't', 'o', ' ', 'b', 'u', 'y', ' ', 's', 'o', 'm', 'e', ' ', 'f', 'l',
'o', 'w', 'e', 'r', 's'
```

我们采用展开的前三批输入和输出，如表 6-1 所示。

表 6-1　前三批输入和输出

Input	Output
'B', 'o', 'b', ' ', 'a'	'o', ' ', 'b', 'a', 'n'
'n', 'd', ' ', 'M', 'a'	'd', ' ', 'M', 'a', 'r'
'r', 'y', ' ', 'w', 'e'	'y', ' ', 'w', 'e', 'n'

通过这样做，RNN 一次看到相对长的数据序列，不像一次处理单个字符。因此，它可以保留更长的序列记忆：

```
train_dataset, train_labels = [],[]
for ui in range(num_unroll):
     train_dataset.append(tf.placeholder(tf.float32,
        shape=[batch_size,in_size],name='train_dataset_%d'%ui))
    train_labels.append(tf.placeholder(tf.float32,
        shape=[batch_size,out_size],name='train_labels_%d'%ui))
```

6.5.3　定义验证数据集

我们将定义一个验证数据集以测量 RNN 随时间的性能情况。我们不会使用验证集中的数据进行训练，只观察验证数据的预测作为 RNN 性能的指标：

```
valid_dataset = tf.placeholder(tf.float32,
       shape=[1,in_size],name='valid_dataset')
valid_labels = tf.placeholder(tf.float32,
      shape=[1,out_size],name='valid_labels')
```

我们使用较长的故事收集验证集，并从最后提取故事的一部分。代码文件中有详细的解释记录，读者可以自行查看。

6.5.4 定义权重值和偏差

在这里，我们将定义 RNN 的几个权重值和偏差参数。

- W_xh：输入和隐藏层之间的权重值。
- W_hh：隐藏层的重复连接的权重值。
- W_hy：隐藏层和输出之间的权重值。

```
W_xh = tf.Variable(tf.truncated_normal(
                        [in_size,hidden],stddev=0.02,
                        dtype=tf.float32),name='W_xh')
W_hh = tf.Variable(tf.truncated_normal([hidden,hidden],
                         stddev=0.02,
                         dtype=tf.float32),name='W_hh')
W_hy = tf.Variable(tf.truncated_normal(
                        [hidden,out_size],stddev=0.02,
                        dtype=tf.float32),name='W_hy')
```

6.5.5 定义状态永久变量

在这里，我们将定义用于区分 RNN 与前馈神经网络的最重要的实体之一：RNN 的状态。状态变量代表 RNN 的记忆。此外，这些被建模为不可训练的 TensorFlow 变量。

我们将首先定义变量（训练数据：prev_train_h 和验证数据：prev_valid_h）以维持用于计算当前隐藏状态的隐藏层的先前状态。我们将定义两个状态变量。一个状态变量在训练期间维持 RNN 的状态，另一个状态变量在验证期间维持 RNN 的状态：

```
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],

dtype=tf.float32),name='train_h',trainable=False)
                        name='prev_h1',trainable=False)
prev_valid_h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
                        name='valid_h',trainable=False)
```

6.5.6 使用展开的输入计算隐藏状态和输出

接下来，我们将定义每个展开输入的隐藏层计算、非标准化分数和预测。为了计算每个隐藏层的输出，我们保存表示每个展开元素的 num_unroll 隐藏状态输出（代码中的 outputs）。然后计算所有 num_unroll 步的非标准化预测（也称为 logits 或得分）和 softmax 预测。

```
# 在 num unroll 步数中，为每个步长添加上计算的 RNN 输出
outputs = list()
# 这将在计算的 num unroll 步数中迭代使用
output h = prev train h
# 计算 num unroll 步数的 RNN 输出(根据截断的 BPTT 的要求)
for ui in range(num unroll):
        output h = tf.nn.tanh(
               tf.matmul(tf.concat([train dataset[ui],output h],1),
                             tf.concat([W xh,W hh],0))
)
outputs.append(output_h)
```

然后计算非标准化预测（y_scores）和标准化预测（y_predictions），如下所示：

```
# 获取我们为 num_unroll 步生成的所有 RNN 输出的分数和预测
y_scores = [tf.matmul(outputs[ui],W_hy) for ui in range(num_unroll)]
y_predictions = [tf.nn.softmax(y_scores[ui]) for ui in range(num_
unroll)]
```

6.5.7 计算损失

在计算预测之后，接着计算 rnn_loss。损失是预测输出和实际输出之间的交叉熵损失。注意，我们调用 tf.control_dependencies(...)函数将 RNN(output_h)的最后一个输出保存到 prev_train_h 变量中。因此，在下一次迭代中，我们可以把先前保存的 RNN 输出作为初始状态：

```
#确保在计算损失之前，使用获得的最后一个 RNN 输出状态来更新状态变量
with tf.control_dependencies([tf.assign(prev_train_h,output_h)]):
       #计算在所有 num_unroll 步中一次性获得的所有预测的 softmax 交叉熵
       rnn_loss = tf.reduce_mean(
                      tf.nn.softmax_cross_entropy_with_logits_v2(
                      logits=tf.concat(y_scores,0),
                      labels=tf.concat(train_labels,0)
        ))
```

6.5.8 在新文本片段的开头重置状态

我们还需要定义隐藏状态重置操作。在测试时生成新的文本块之前，尤其需要使用重置操作，否则 RNN 将继续产生依赖于先前产生的文本，从而导致高度相关的输出。若是如此，就很糟糕，因为它最终将引起 RNN 一遍又一遍地输出相同的词。实际上，在训练期间重置状态是否有益仍然存在争议。不过，我们为此定义了 TensorFlow 操作：

```
# 重置隐藏状态
```

```
reset_train_h_op = tf.assign(prev_train_h,tf.zeros(
                                    [batch_size,hidden],
                                    dtype=tf.float32))
reset_valid_h_op = tf.assign(prev_valid_h,tf.zeros(
                                    [1,hidden],dtype=tf.float32))
```

6.5.9 计算验证输出

这里，类似于训练状态、损失和预测计算，我们定义了用于验证的状态、损失和预测：

```
#计算下一个有效状态（仅 1 步）
next_valid_state = tf.nn.tanh(tf.matmul(valid_dataset,W_xh) +
                                   tf.matmul(prev_valid_h,W_hh))
# 使用 RNN 的状态输出计算预测
# 但在此之前，将 RNN 的最新状态输出分配给验证阶段的状态变量
# 所以需要确保执行 valid_predictions 操作以更新验证状态
with tf.control_dependencies([tf.assign(prev_valid_h,next_valid_
state)]):
          valid_scores = tf.matmul(next_valid_state,W_hy)
          valid_predictions = tf.nn.softmax(valid_scores)
```

6.5.10 计算梯度和优化

由于已经定义了 RNN 的损失，我们将使用随机梯度方法来计算梯度并使用它们。为此，我们使用 TBPTT。在这种方法中，我们将随时间而展开 RNN（类似于我们如何随时间展开输入）并计算梯度，然后回滚计算的梯度以更新 RNN 的权重值。此外，我们将使用 AdamOptimizer，这是一种基于动量的优化方法，它显示出比标准随机梯度下降更好的收敛速度。此外，使用 Adam Optimizer 时请确保使用较小的学习率（例如介于 0.001 和 0.0001 之间）。我们还将使用梯度裁剪来防止任何潜在的梯度爆炸：

```
rnn_optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
gradients, v = zip(*rnn_optimizer.compute_gradients(rnn_loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
rnn_optimizer = rnn_optimizer.apply_gradients(zip(gradients, v))
```

6.6 输出新生成的文本片段

下面来看如何使用训练模型输出新文本。在这里，我们将预测一个词，将该词用作下一个输入并预测另一个词，以这种方式继续进行几个时间步长：

```
#获取测试阶段隐藏节点的先前状态
```

```
prev test h = tf.Variable(tf.zeros([test batch size,hidden],
                                    dtype=tf.float32),name='test h')
# 测试数据集
test dataset = tf.placeholder(tf.float32, shape=[test batch size,
                                          in size],name='test dataset')
# 计算测试数据的隐藏输出
next test state = tf.nn.tanh(tf.matmul(test dataset,W xh) +
                                        tf.matmul(prev test h,W hh)
                            )
# 确保每次进行预测时都更新测试阶段的隐藏状态
with tf.control_dependencies([tf.assign(prev_test_h,next_test_
state)]):
    test_prediction = tf.nn.softmax(tf.matmul(next_test_state,W_hy))
# 请注意，我们在重置测试状态时使用了小的赋值（一些小的填充（imputations））
# 这有助于为生成的文本添加更多变量
reset_test_h_op = tf.assign(prev_test_h,tf.truncated_normal(
                                       [test_batch_size,hidden],stddev=0.01,
                                        dtype=tf.float32))
```

6.7 评估 RNN 的文本结果输出

这里我们将显示使用 RNN 生成的文本片段，显示不使用输入展开时以及使用输入展开时的结果。

在没有输入展开的情况下，在 10 个 epoch（即训练 10 遍）之后，得到以下结果：

```
him he had then the king then the king ferdinand the fait, and the king, and the
king, and the king, and the king, and the king, and the king, and the king, and
the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and the king,
and the king, and the king, and the king, and the king, and the king, and...
```

而在输入展开时，在 10 个 epoch 后将获得以下内容：

```
... god grant that our sister may be here, and then we shall be free.
when the maiden,who was standing behind the door watching, heard thatwish,
```

```
she came forth, and on this all the ravens were restored to their
human form again. and they embraced and kissed each other,
and went joyfully home whome, and wanted to eat and drink, and
looked for their little plates and glasses. then said one after
the other, who has eaten something from my plate. who has drunk
out of my little glass. it was a human mouth. and when the
seventh came to the bottom of the glass, the ring rolled against
his mouth. then he looked at it, and saw that it was a ring
belonging to his father and mother, and said, god grant that our
sister may be here, and then we shall be free. …
```

从这些结果中，我们可以观察到一点，与每次处理单个输入相比，它实际上有助于随时间推移进行输入展开。然而即使展开输入，也存在一些语法错误和罕见的拼写错误（这是可以接受的，因为我们一次处理两个字符）。

另一点值得注意的是，我们的 RNN 试图通过组合之前看到的不同故事来产生一个新的故事。我们可以看到，首先谈论的是乌鸦，然后通过谈论盘子和有人从盘子里吃东西，把故事转移到类似于金发姑娘和三只熊（Goldilocks and the Three Bears）的故事。接下来，这个故事引出了一枚戒指。

这意味着 RNN 已经学会了组合故事，并想出新的故事。然而，我们可以通过引入更好的学习模型（例如 LSTM）和更好的搜索技术（例如集束搜索）来进一步改善这些结果，这些将在下一章中介绍。

提示

由于语言的复杂性和 RNN 较小的表示能力，在整个学习过程中，不太可能获得与本文所示的文本一样良好的输出。因此，我们挑选了一些生成的文本，以表达我们的观点。

注意，这是一个精心挑选的文本生成示例，仔细观察会发现，随着时间的推移，如果继续多次迭代来进行预测，RNN 会一遍又一遍地重复同一块文本。我们可以看到，这在前面的文本片段中已经存在，其中第一个句子与最后一个句子相同。随着数据集大小的不断增加，我们会发现，这个问题会变得更加突出。这是由于梯度消失问题导致我们的 RNN 的记忆能力不足引起的。而为了减少这种影响，在 6.9 和 6.10 节中，我们将讨论 RNN 的一种变体——具有上下文特征的 RNN（RNN-CF），可以减少这种影响。

6.8 困惑度——文本生成结果质量的度量

在信息论中，困惑度（Perplexity）用来度量一个概率分布模型或概率模型预测样本的好坏程度，也可以用来比较两个概率分布或概率模型。低困惑度的概率分布模型或概率模型能更好地预测样本。关于困惑度的更多信息，读者可以查看维基百科给出的解释（https://en.wikipedia.org/wiki/Perplexity）。

我们的工作不仅仅是利用模型生成新的文本，还需要利用一个方法来衡量生成文本的质量。一

种方法是测量 RNN 遇到给定输入的输出会产生多大程度的困惑程度（Perplexed，或者惊讶程度（Surprised））。也就是说，如果输入x_i及其对应的输出y_i的交叉熵损失是$l(x_i, y_i)$，那么困惑度将是：

$$p(x_i, y_i) = e^{l(x_i, y_i)}$$

使用这个方法可以计算大小为 N 的训练数据集的平均困惑，具体如下：

$$p(D_{train}) = (1/N)\sum_{(i=1)}^{N} p(x_i, y_i)$$

在图 6-14 中，我们展示了训练困惑度和验证困惑度随时间的变化情况。我们可以看到，训练难度随着时间的推移而稳步下降，其中验证困惑度则呈显著波动状态。其实这是可以理解的，因为在验证困惑度中，本质上评估的是 RNN 基于我们对训练数据的学习来预测不可见文本的能力。由于语言任务很难建模，因此这是一个非常困难的任务，出现这些波动是很正常的事情。

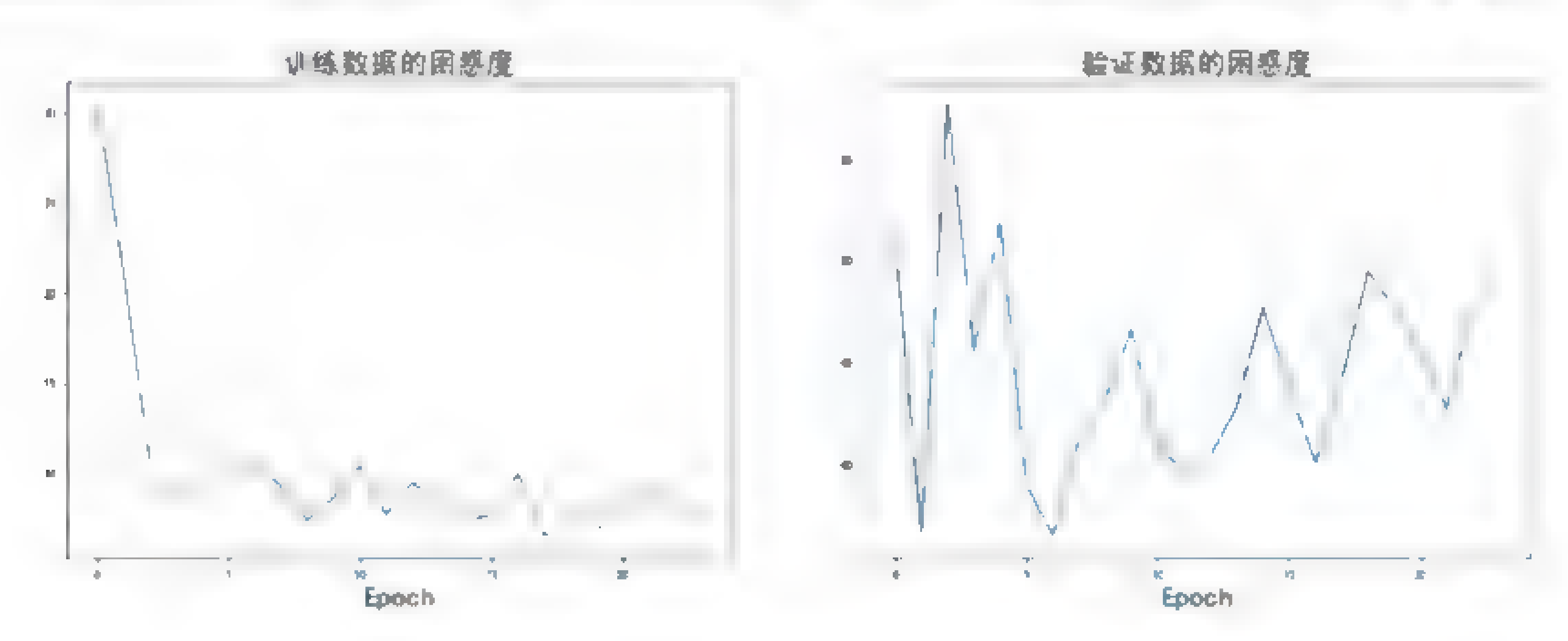


图 6-14 训练数据和验证数据的困惑度随时间变化的示意图

这里有一种改善结果的方法，就是向 RNN 增加更多的隐藏层，因为更深层次的模型通常可以提供更好的结果。我们在 ch6 文件夹中的 6_rnn_language_bigram_multilayer.ipynb 中实现了一个三层 RNN，读者可以自行查看。

现在我们来看一个问题，是否有更好的 RNN 的变体会工作得更好呢？例如，有没有 RNN 的变体能更有效地解决梯度消失的问题？在下一节中，我们来讨论一个称为 RNN-CF 的变体。

6.9 具有上下文特征的循环神经网络——RNN-CF

前面我们讨论了训练简单 RNN 的两个重要挑战：梯度爆炸和梯度消失。我们知道可以用一个简单的策略来防止梯度爆炸，例如梯度裁剪，从而实现更稳定的训练。然而，对于梯度消失问题而言，还需要更加努力地去做，因为没有简单的缩放/裁剪机制可以解决梯度消失问题，就像我们对梯度爆炸所做的那样。因此，我们需要修改 RNN 本身的结构，明确地赋予它记忆数据序列中较长模式的能力。Tomas Mikolov 等人于 2015 年在其文章《*Learning Longer Memory in Recurrent Neural Networks*》（International Conference on Learning Representations）中提出了 RNN-CF，是标准 RNN

修改后的一种模型，能够帮助 RNN 更长时间地记住数据序列中较长的模式。

RNN-CF 通过引入新的状态和一组新的前向和循环连接来减少梯度消失。换言之，与仅具有单个状态向量的标准 RNN 相比，RNN-CF 将具有两个状态向量。其主要思想是，一个状态向量变化缓慢，用于保留较长的记忆，而另一个状态向量可以快速变化，用于短期记忆工作。

6.9.1 RNN-CF 的技术说明

在这里，我们用几个参数来调整传统的 RNN，以帮助维持更长时间的记忆。除了标准 RNN 模型中存在的常规状态向量之外，这些调整还包括引入新的状态向量。因此，还引入了几个前向和循环的权重值集。在抽象层次上，图 6-15 给出了 RNN 与 RNN-CF 比较的示意图。

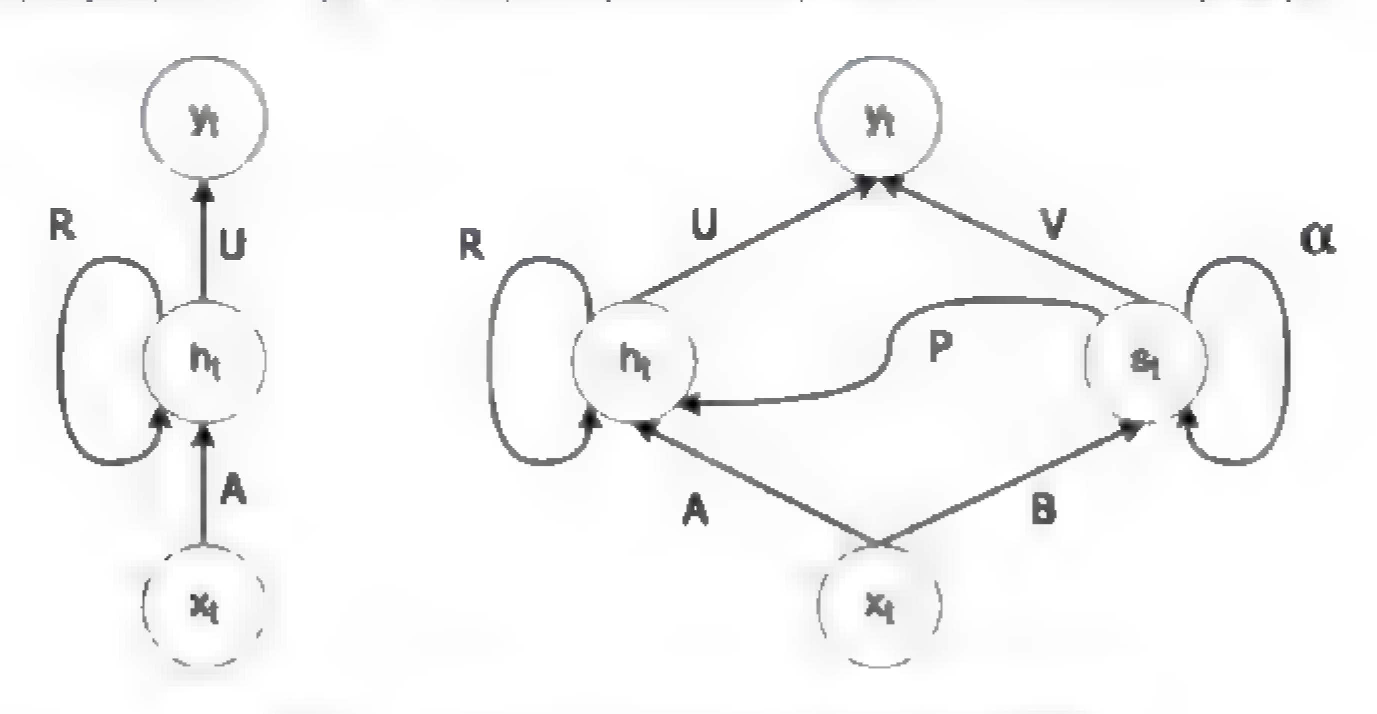


图 6-15 RNN 与 RNN-CF 比较的示意图

从图 6-15 中可以看到，与传统的 RNN 相比，RNN-CF 具有一些额外的权重值。现在让我们仔细看看这些图层和权重值是做什么的。

首先，输入由两个隐藏层接收，就像在 RNN 中发现的传统隐藏层一样。我们已经看到，仅使用这个隐藏层在保留长期记忆方面效果不佳。但是，我们可以通过强制循环矩阵接近同一性并消除非线性来强制隐藏层更长时间地保留记忆。当循环矩阵接近同一性而没有非线性时，发生在 h 上的任何变化都应该始终来自于输入的变化。换句话说，先前的状态对改变当前状态的影响较小。这导致状态变化慢于密集的权重值矩阵和非线性。因此，这种状态有助于更长时间地保留记忆。支持循环矩阵接近 1 的另一个原因是，当权重值接近 1 时，出现在推导中的项如$w^{(n-1)}$将不会消失或爆炸。但是，如果我们仅使用此而没有非线性的隐藏层的话，梯度就永远不会减少。在这里，对于梯度减小的影响，旧的输入应该远小于最近的输入。然后，我们需要通过时间将梯度传播到输入的起始位置，但这样的成本很高。因此，为了充分利用这两个方面，我们保留了这两个层：可以快速变化的标准 RNN 状态层(h_t)，以及变化更慢的上下文特征层(s_t)。这个新层称为上下文层(Context Layer)，是一个有助于保持长期记忆的层。RNN-CF 的更新规则如下（这里我们没有看到$s_{(t-1)}$被一个单位矩阵所乘，是因为$Is_{(t-1)}= s_{(t-1)}$）。

$$s_t = (1-\alpha)Bx_t + \alpha s_{(t-1)}$$
$$h_t = \sigma(Ps_t + Ax_t + Rh_{(t-1)})$$
$$y_t = softmax(Uh_t + Vs_t)$$

与 RNN-CF 相关的符号总结在表 6-2 中。

表 6-2 RNN-CF 相关的符号列表

标记符	具体含义
x_t	当前的输入
h_t	当前的状态向量
y_t	当前的输出
s_t	当前上下文的特征向量
A	x_t和h_t之间的权重矩阵
B	x_t和s_t之间的权重矩阵
R	h_t的循环连接
α	控制$s_{(t-1)}$对s_t影响的常量
P	h_t和s_t之间的权重矩阵
U	h_t和y_t之间的权重矩阵
V	s_t和y_t之间的权重矩阵

6.9.2 RNN-CF 的实现

前面我们已经讨论了 RNN-CF 包含的额外的状态向量以及它如何帮助防止梯度消失。这里我们将讨论 RNN-CF 的实现。除了传统 RNN 实现中的隐藏层（h_t）、W_{xh}（表中的 A）、W_{hh}（表中的 R）和W_{hy}（表中的 U）之外，现在还需要另外三组权重值，即我们将定义 B、P 和 V。此外，我们将定义一个新变量来包含s_t（Hidden_Context，不包括h_t）。

6.9.3 定义 RNN-CF 超参数

首先我们将定义超参数，包括之前定义的超参数和新参数。一个新的超参数定义了上下文特征层中的神经元数量s_t，其中 alpha 表示等式中的 α。

```
hidden_context = 64
alpha = 0.9
```

6.9.4 定义输入和输出占位符

正如我们对标准 RNN 所做的那样，定义占位符以包含训练输入和输出、验证输入和输出以及测试输入：

```
# 训练数据集
train_dataset, train_labels = [],[]
for ui in range(num_unroll):
    train_dataset.append(tf.placeholder(tf.float32,
                                        shape=[batch_size,in_size],
```

```
                                 name 'train dataset %d'%ui))
train labels.append(tf.placeholder(tf.float32,
                          shape=[batch size,out size],
                          name='train labels %d'%ui))

# 验证数据集
valid dataset = tf.placeholder(tf.float32,
                                 shape=[1,in_size],name='valid dataset')
valid labels = tf.placeholder(tf.float32,
                                shape=[1,out_size],name='valid_labels')
# 测试数据集
test_dataset = tf.placeholder(tf.float32,
                               shape=[test_batch_size,in_size],
                               name='save_test_dataset')
```

6.9.5 定义 RNN-CF 的权重值

我们将定义 RNN-CF 计算所需的权重值。正如在符号表中看到的那样，这里 RNN-CF 计算需要 6 组权重值（A、B、R、P、U 和 V）。而在传统的 RNN 计算中，只需要三组权重值就可以了。

```
# 输入层和隐藏层(h)之间的权重值
A = tf.Variable(tf.truncated_normal([in_size,hidden],
                      stddev=0.02,dtype=tf.float32),name='W_xh')
B = tf.Variable(tf.truncated_normal([in_size,hidden_context],
                      stddev=0.02,dtype=tf.float32),name='W_xs')

# 隐藏层(h)间的权重值
R = tf.Variable(tf.truncated_normal([hidden,hidden],
                      stddev=0.02,dtype=tf.float32),name='W_hh')
P = tf.Variable(tf.truncated_normal([hidden_context,hidden],
                      stddev=0.02,dtype=tf.float32),name='W_ss')

# 隐藏层(h)和输出层（y）之间的权重值
U = tf.Variable(tf.truncated_normal([hidden,out_size],
                      stddev=0.02,dtype=tf.float32),name='W_hy')
V = tf.Variable(tf.truncated_normal([hidden_context,out_size],stddev=0.02,
                                                   dtype=tf.float32),
   name='W_sy')
# 训练数据的状态变量
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],dtype=tf.float32),
                                name='train_h',trainable=False)

prev train s = tf.Variable(tf.zeros([batch size,hidden context],
```

```
                                        dtype tf.float32),name 'train s',
                                        trainable False)

# 验证数据的状态变量
prev valid h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
                                         name='valid h',trainable=False)
prev valid s = tf.Variable(tf.zeros([1,hidden context],dtype=tf.float32),
                                        name='valid s',trainable=False)
#  测试数据的状态变量
prev_test_h = tf.Variable(tf.zeros([test_batch_size,hidden],
                                       dtype=tf.float32),
                                       name='test_h')
prev_test_s = tf.Variable(tf.zeros([test_batch_size,hidden_context],
                                      dtype=tf.float32),name='test_s')
```

6.9.6　用于维护隐藏层和上下文状态的变量和操作

我们将定义 RNN-CF 的状态变量。除了传统 RNN 中的h_t之外，我们还需要对上下文特征有一个单独的状态，即s_t。总之，我们将有 6 个状态变量。其中，三个状态变量在训练、验证和测试期间维持状态向量h_t，而另外三个状态变量在训练、验证和测试期间维持状态向量s_t：

```
# 训练数据的状态变量
prev_train_h = tf.Variable(tf.zeros([batch_size,hidden],
                                      dtype=tf.float32),
                                      name='train_h',trainable=False)
prev_train_s = tf.Variable(tf.zeros([batch_size,hidden_context],
                                      dtype=tf.float32),name='train_s',
                                      trainable=False)

# 验证数据的状态变量
prev_valid_h = tf.Variable(tf.zeros([1,hidden],dtype=tf.float32),
                                       name='valid_h',trainable=False)
prev_valid_s = tf.Variable(tf.zeros([1,hidden_context],
                                       dtype=tf.float32),
                                       name='valid_s',trainable=False)
# 测试数据的状态变量
prev_test_h = tf.Variable(tf.zeros([test_batch_size,hidden],
                                    dtype=tf.float32),
                                    name='test_h')
prev_test_s = tf.Variable(tf.zeros([test_batch_size,hidden_context],
                                    dtype-tf.float32),name 'test_s')
```

接下来，我们定义所需的重置操作，以重置状态：

```
reset_prev_train_h_op = tf.assign(prev_train_h,tf.zeros([batch_size,
                                  hidden], dtype=tf.float32))
reset_prev_train_s_op = tf.assign(prev_train_s,tf.zeros([batch_size,
                                 hidden_context],dtype=tf.float32))
reset_valid_h_op = tf.assign(prev_valid_h,tf.zeros([1,hidden],
                          dtype=tf.float32))
reset_valid_s_op = tf.assign(prev_valid_s,tf.zeros([1,hidden_context],
                          dtype=tf.float32))
# 加入噪声来估算测试状态
reset_test_h_op = tf.assign(prev_test_h,tf.truncated_normal(
                                      [test_batch_size,hidden],
                                      stddev=0.01,
                                      dtype=tf.float32))
reset_test_s_op = tf.assign(prev_test_s,tf.truncated_normal(
                                     [test_batch_size,hidden_context],
                                     stddev=0.01,dtype=tf.float32))
```

6.9.7 计算输出

在定义了所有输入、变量和状态向量之后，现在根据前面提到的方程式来计算 RNN-CF 的输出。首先，将状态向量初始化为零。其次，将针对一组固定的时间步长（根据 BPTT 的需要）展开我们的输入，并分别计算每个展开步骤的非标准化输出（有时称为 logits 或得分）。接着，将连接属于每个展开时间步长的所有 y 值，最后计算所有这些项的平均损失，并将其与真实标签进行比较：

```
# 训练得分（非标准化）值和预测（标准化）
y_scores, y_predictions = [],[]

#这些将在 num_unroll 计算步骤中迭代使用
next_h_state = prev_train_h
next_s_state = prev_train_s

# 在 num_unroll 步数中为每个步长添加计算的 RNN 输出
next_h_states_unrolled, next_s_states_unrolled = [],[]

#为 num_unroll 步数计算 RNN 的输出（根据截断的 BPTT 的要求）
for ui in range(num_unroll):
    next_h_state = tf.nn.tanh(
        tf.matmul(tf.concat([train_dataset[ui],prev_train_h,
                     prev_train_s],1),
                     tf.concat([A,R,P],0))
   )
next_s_state = (1-alpha)*tf.matmul(train_dataset[ui],B) +
```

```
                    alpha * next_s_state
    next_h_states_unrolled.append(next_h_state)
    next_s_states_unrolled.append(next_s_state)

#获取为 num_unroll 步生成的所有 RNN 输出的分数和预测
y_scores = [tf.matmul(next_h_states_unrolled[ui],U) +
                tf.matmul(next_s_states_unrolled[ui],V)
               for ui in range(num_unroll)]
y_predictions = [tf.nn.softmax(y_scores[ui]) for ui in range(num_
unroll)]
```

6.9.8　计算损失

本节定义有关 RNN-CF 的损失的计算。此操作与之前我们为标准 RNN 定义的操作相同，如下所示：

```
# 确保在计算损失之前，使用我们获得的最后一个 RNN 的输出状态来更新状态变量
with tf.control_dependencies([tf.assign(prev_train_s, next_s_state),
                              tf.assign(prev_train_h,next_h_state)]):
      rnn_loss = tf.reduce_mean(
                  tf.nn.softmax_cross_entropy_with_logits_v2(
                  logits=tf.concat(y_scores,0),
                  labels=tf.concat(train_labels,0) ))
```

6.9.9　计算验证输出

与在训练时计算的输出类似，我们这里计算验证输入的输出。但是，我们不会像处理训练数据那样展开输入，因为在预测期间不需要展开：

```
# 验证数据相关的推理逻辑（非常类似训练推理逻辑）
# 计算下一个验证状态（仅 1 步）
next_valid_s_state = (1-alpha) * tf.matmul(valid_dataset,B) +
                             alpha * prev_valid_s
next_valid_h_state = tf.nn.tanh(tf.matmul(valid_dataset,A) +
                                tf.matmul(prev_valid_s, P) +
                                tf.matmul(prev_valid_h,R))

#使用 RNN 的状态输出计算预测
# 但在此之前，将 RNN 的最新状态输出分配给验证阶段的状态变量
# 因此，需要确保执行 rnn_valid_loss 操作以更新验证状态
with tf.control_dependencies([tf.assign(prev_valid_s, next_valid_s_state),
                              tf.assign(prev_valid_h,next_valid_h_state)]):
```

```
valid_scores = tf.matmul(prev_valid_h, U) + tf.matmul(prev_valid_s, V)
valid_predictions = tf.nn.softmax(valid_scores)
```

6.9.10 计算测试输出

我们现在可以定义输出计算以生成新的测试数据：

```
# 测试数据相关的推理逻辑
#计算测试数据的隐藏输出
next_test_s = (1-alpha)*tf.matmul(test_dataset,B)+ alpha*prev_test_s

next_test_h = tf.nn.tanh(
        tf.matmul(test_dataset,A) + tf.matmul(prev_test_s,P) +
        tf.matmul(prev_test_h, R) )
# 确保每次进行预测时都已更新了测试阶段的隐藏状态
with tf.control_dependencies([tf.assign(prev_test_s,next_test_s),
                              tf.assign(prev_test_h,next_test_h)]):
       test_prediction = tf.nn.softmax(
                        tf.matmul(prev_test_h,U) + tf.matmul(prev_test_s,V)
)
```

6.9.11 计算梯度和优化

下面使用优化器来最小化损失，这与之前对于传统 RNN 所做的方法相同：

```
rnn_optimizer = tf.train.AdamOptimizer(learning_rate=.001)
gradients, v = zip(*rnn_optimizer.compute_gradients(rnn_loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
rnn_optimizer = rnn_optimizer.apply_gradients(zip(gradients, v))
```

6.10 使用 RNN-CF 生成的文本

这里我们将对 RNN 和 RNN-CF 生成的文本进行定性和定量的比较。首先比较使用 20 个训练文档获得的结果。之后将训练文档的数量提高到 100，以查看 RNN 和 RNN-CF 是否能够合并大量数据，以便输出质量更好的文本。

首先，对于使用 20 个训练文档，RNN-CF 生成的文本如下：

```
the king's daughter, who had
no more excuses left to make. they cut the could not off, and her his
first rays of life in the garden,
and was amazed to see with the showed to the grown mighted and the
```

```
seart the answer to star's brothers, and seeking the golden apple, we
flew over the tree to the seadow where her
heard that he could not have discome.

emptied him by him. she himself 'well i ston the fire struck it was
and said the youth, farm of them into the showed to shudder, but here
and said the fire himself 'if i could but the youth, and thought that
is that shudder.'
'then, said he said 'i will by you are you, you.' then the king, who
you are your
wedding-mantle. you are you are you
bird in wretch me. ah. what man caller streep them if i will bed.
the youth
begged for a hearing, and said 'if you will below in you to be your
wedding-mantle.' 'what.' said he, 'i shall said 'if i hall by you are you
bidden it i could not have
```

就文本的质量而言，与标准的 RNN 相比，我们看不到太大的差别。至于为什么 RNN-CF 没有比标准 RNN 表现得更好，Mikolov 等人在他们的论文《*Learning Longer Memory in Recurrent Neural Network*》中提到："当标准隐藏单元的数量足以捕获短期模式时，学习自循环权重值似乎就不再重要了"。

因此，如果隐藏单元的数量足够大，RNN-CF 与标准 RNN 相比就没有明显的优势。这可能就是我们观察到这一点的原因所在。在代码中，我们只使用了 64 个隐藏的神经元和一个相对较小的语料库，它足以表示一个故事，达到 RNN 所能达到的水平。

我们再来看看增加数据量是否真的有助于 RNN-CF 执行得更好。对于这里的示例，在训练了大约 50 个 epoch 之后，我们将把文档数量增加到 100 个。

以下是标准 RNN 的输出：

```
 they were their dearest and she she told him to stop crying to the
king's son they were their dearest and she she told him to stop crying
to the king's son they were their dearest and she she told him to stop
crying to the king's son they were their dearest and she she told him
to stop crying to the king's son they were their dearest and she she
told him to stop
```

我们可以看到，与使用较少数据时的表现相比，RNN 变得更糟了。数据量大和模型容量不足对标准 RNN 有不利影响，导致它们输出低质量的文本。

以下是 RNN-CF 的输出。我们可以看到，就变化而言，RNN-CF 比标准 RNN 做得好得多：

```
then they could be the world. not was now from the first for a set
out of his pocket, what is the world. then they were all they were
forest, and the never yet not rething, and took the
children in themselver to peard, and then the first her. then the was
```

```
in the first, and that he was to the first, and that he was to the
kitchen, and said, and had took the
children in the mountain, and they were hansel of the fire, gretel of
they were all the fire, goggle-eyes and all in the moster. when she
had took the changeling the little elves, and now ran into them, and she bridge
away with the witch form,
and their father's daughter was that had neep himselver in the horse,
and now they lived them himselver to them, and they were am the
marriage was all they were and all of the marriage was anger of the
forest, and the manikin was laughing, who had said they had not know,
and took the children in themselver to themselver and they lived them himselver
to them
```

因此，当数据充足时，RNN-CF 模型实际上优于标准 RNN 模型。我们将为这两个模型绘制随时间变化的训练和验证的困惑度。正如我们所看到的，在训练困惑度方面，RNN-CF 和标准 RNN 没有显著差异。最后，在验证困惑度图（参见图 6-16）中可以看到，与标准 RNN 相比，RNN-CF 显示了更少的波动。

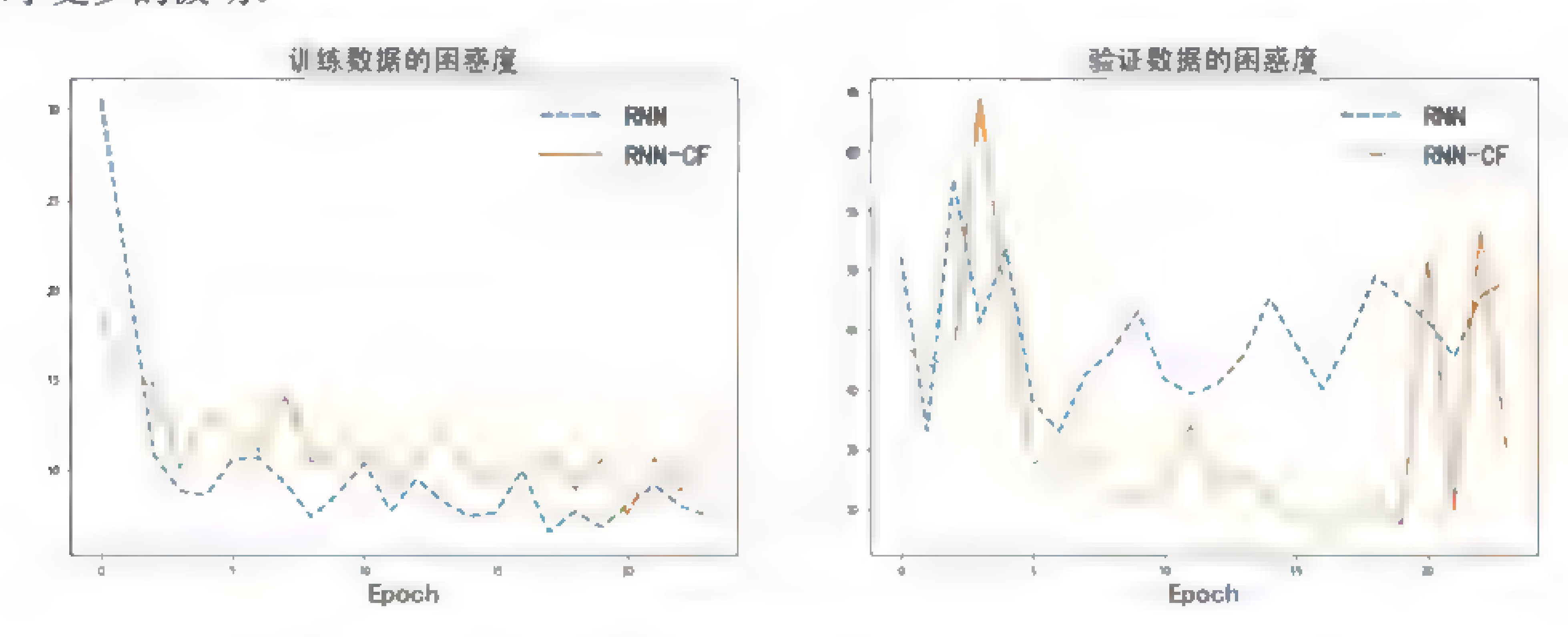


图 6-16 RNN、RNN-CF 的训练数据困惑度和验证数据困惑度示意图

在这里可以得出一个重要的结论，当数据量较小时，标准 RNN 可能是过拟合数据。也就是说，RNN 可能按原样记忆数据，而不是试图学习数据中存在的更常见的模式。当 RNN 被大量数据淹没并且被训练更长时间（比如大约训练 50 遍，即 50 个 epoch）时，这种弱点变得更加突出。产生的文本质量下降，验证困惑度的波动也比较大。而 RNN-CF 对小量数据和大量数据均表现出一定的一致性。

6.11 总结

在本章中，我们首先通过计算图及其展开知道了每个节点表示在某个时刻 t 的状态，函数 f 可以将 t 处的状态映射到 t + 1 处的状态，而在有外部信号 x_t 驱动的情况下，涉及循环的任何函数本质上可以被认为是一种循环神经网络，其中循环神经网络隐藏单元的值可以被定义（通过式

（6.2）），而且我们对于展开图和循环图做了细分层面的解读。

其次，对于 RNN 的解读部分，我们详细介绍了序列数据模型和数学层面的内容，通过相关公式和图示对于 RNN 的内在机制逐步进行了展开和剖析，并在数学层面进行了详细解读。

接着，我们介绍了通过时间的反向传播算法（BPTT）。在这部分，我们认识了反向传播的工作原理、为什么不能对 RNN 使用标准反向传播、如何使用 BPTT 对数据进行 RNN 训练、截断 BPTT 和 BPTT 的局限性等，解读了其局限性中的常见问题：梯度下降和梯度爆炸，并给出了与之对应的解决方法。

然后，我们继续研究 RNN 的实际应用，讨论了 4 种主要类型的 RNN。一对一结构用于诸如文本生成、场景分类和视频帧标记之类的任务。多对一结构用于情感分析，逐字处理句子/短语（与单一处理完整句子相比）。一对多结构在图像字幕任务中很常见，将单个图像映射到描述图像的任意长句子短语。利用多对多结构进行机器翻译任务。

接下来，我们介绍了一个有趣的 RNN 应用——文本生成。我们使用童话语料库来训练 RNN，特别是将故事中的文本打破为 bigrams（一个二元组包含两个字符）。我们通过给出一组从故事中选择的二元组作为输入和随后的双字母（来自输入）作为输出来训练 RNN。然后通过最大化正确预测下一个二元组的准确性来优化 RNN。按照这个程序，我们要求 RNN 生成一个不同的故事，对生成的结果做了两个重要的观察：

- 实际上，随着时间的推移，展开输入有助于更长时间地保持记忆。
- 即使展开，RNN 也只能存储有限数量的长期记忆。

因此，我们研究了一种能够捕获更长记忆的 RNN 变体，被称为 RNN-CF。RNN-CF 具有两个不同的层：隐藏层（在简单 RNN 中找到的传统隐藏层）和上下文层（用于持久记忆、长期记忆）。我们看到，当与小数据集一起使用时，使用这个额外的上下文层并没有明显的帮助，因为在 RNN 中有一个相当复杂的隐藏层，但是当使用更多数据时，它产生了稍好的结果。

在下一章中，我们将讨论一种更强大的 RNN 模型，被称为长短期记忆（LSTM）网络，可进一步减少梯度消失的不利影响，从而产生更好的结果。

第 7 章

长短期记忆

在上一章中，我们介绍了循环神经网络，了解到在训练 RNN 时可能会遇到梯度不稳定（消失和爆炸）的问题，导致我们的神经网络学到的东西不符合预期要求。然而幸运的是，我们可以引入一个称为长短期记忆（Long Short-Term Memory，LSTM）的块（Block，也称为 Cell，即“细胞”）进入 RNN 中。LSTM 由 Sepp Hochreiter 和 JürgenSchmidhuber 于 1997 年提出，并于 2000 年由 Felix Gers 的团队进行了改进，它属于 RNN 的一种改进类型，非常适合处理时间序列数据。这种类型的神经网络最近在深度学习的背景下被重新发现，因为它没有梯度消失的问题，并提供了出色的结果和性能。基于 LSTM 的网络是时间序列预测和分类的理想选择，并且正在取代许多传统的深度学习方法。

由于梯度消失会阻止模型学习的长期依赖性，而在时间序列中的重要事件之间可能存在未知持续时间的滞后，面对这种困境，我们对 LSTM 进行了精心设计，能够避免之前的梯度不稳定问题，使得 LSTM 能够存储比普通 RNN 更长的记忆（数百个时间步长）。在模型训练中，RNN 仅保持单个隐藏状态，而 LSTM 具有更多的参数，可以做到该存储的存储、该丢弃的丢弃，而 RNN 却无法决定存储哪些信息以及丢弃哪些信息，因为在每个训练步骤中都强制要更新隐藏状态。对于间隙长度的相对不敏感性是 LSTM 相对于 RNN、隐马尔可夫模型和其他序列学习模型在许多应用中的优势所在，这种优势尤其在海量数据样本的可用性上已经得到很好的验证，在许多序列任务（例如语言建模、股票预测、机器翻译等）上都有优异的表现。

本章中将首先介绍 LSTM 对长期依赖性如何进行存储（或称为记忆）。其次，对于 LSTM 的数学基础框架进行解读，并用一个小示例加以细化解释。然后介绍专门为改进标准 LSTM 产生预测而引入的几种技术，还将介绍双向 LSTM 模型。最后讨论 LSTM 的两个著名变体：Peephole 连接和门控循环单元（GRU）。

7.1　LSTM 简述

LSTM 本质上是一种更高级的 RNN，可以学习长期依赖信息，在解决很多问题上都表现了优异的性能，这与它的内部结构有很大的关系，其整体结构图如图 7-1 所示。

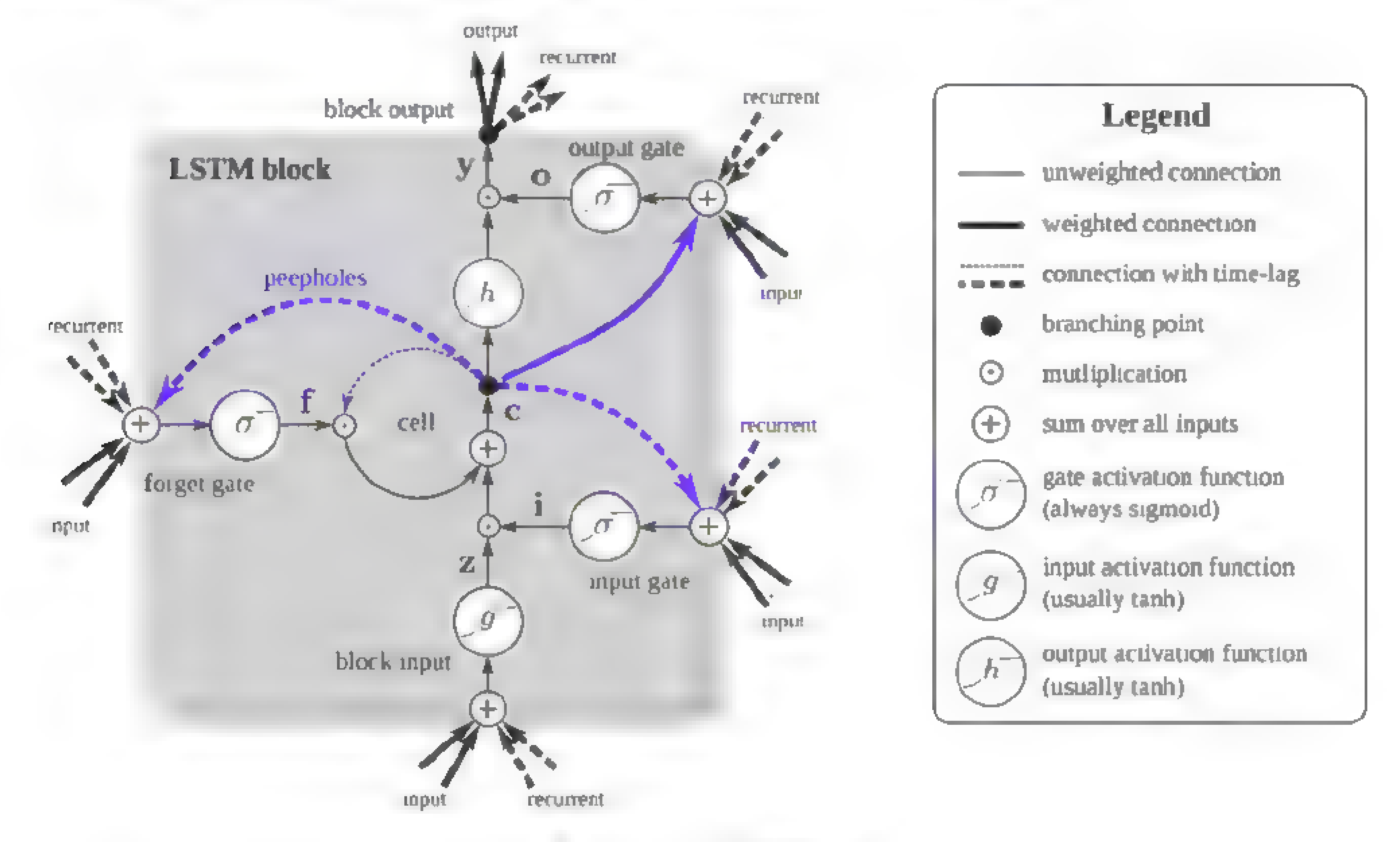


图 7-1　LSTM 整体结构

由图 7-1 可知，LSTM 的整体结构如下：

（1）5 个主要构件，分别如下。

①细胞状态（Cell State）：LSTM 细胞的内部状态（记忆）。

②隐藏状态（Hidden State）：用于计算预测的外部隐藏状态。

③输入门（Input Gate）：决定了当前输入被读入细胞状态的程度。

④遗忘门（Forget Gate）：确定了将先前的细胞状态发送到当前细胞状态的程度。

⑤输出门（Output Gate）：决定了将多少细胞状态输出到隐藏状态。

（2）特殊的神经元结构，包含 4 个 Input（3 个 Gate 控制信号以及输入数据）和 1 个 Output。

（3）激活函数通常选用 Sigmoid 函数（也叫 Logistic 函数），个别使用 tanh 函数。Sigmoid 激活函数用于将输出压缩到 0 和 1 之间，用于表示 Gate 的打开程度。

为了方便介绍 LSTM 的细胞架构，这里给出 LSTM 中数据流的抽象图，如图 7-2 所示。正如图 7-2 中所示，细胞将输出某个状态，该状态依赖于（具有非线性激活函数）之前的状态和当前的输入。但是，在 RNN 中，细胞状态总是随着接下来的每个输入而变动的，这种现象的存在对于存储长期依赖性是非常不利的。

这里需要强调的是，LSTM 可以决定何时替换、更新或遗忘存储在细胞状态中的每个神经元内

的信息。换句话说，如果有需要的话，LSTM 本身其实存在一种机制（其实就是门控机制）来保持细胞状态不变，使它们能够存储长期依赖性——即保持长期记忆。

LSTM 拥有细胞需要执行每个操作的各类门（Gate），而这些门的打开程度在 0 和 1 之间（通常是 Sigmoid 函数）是连续的，其中 0 表示该门处于闭合状态，没有信息流可以通过该门，1 表示该门处于打开状态，所有信息都可以通过该门。LSTM 的这些功能体现在图 7-2 中。每个门决定各种数据（例如当前输入状态、先前隐藏状态或先前的细胞状态）有多少流入状态（最终隐藏状态或细胞状态），每条线的粗细表示从该门流出的信息量大小。例如，输入门相比于先前最终隐藏状态更喜欢允许更多的当前输入通过，而遗忘门则更喜欢让更多的先前最终隐藏状态而不是当前输入通过。

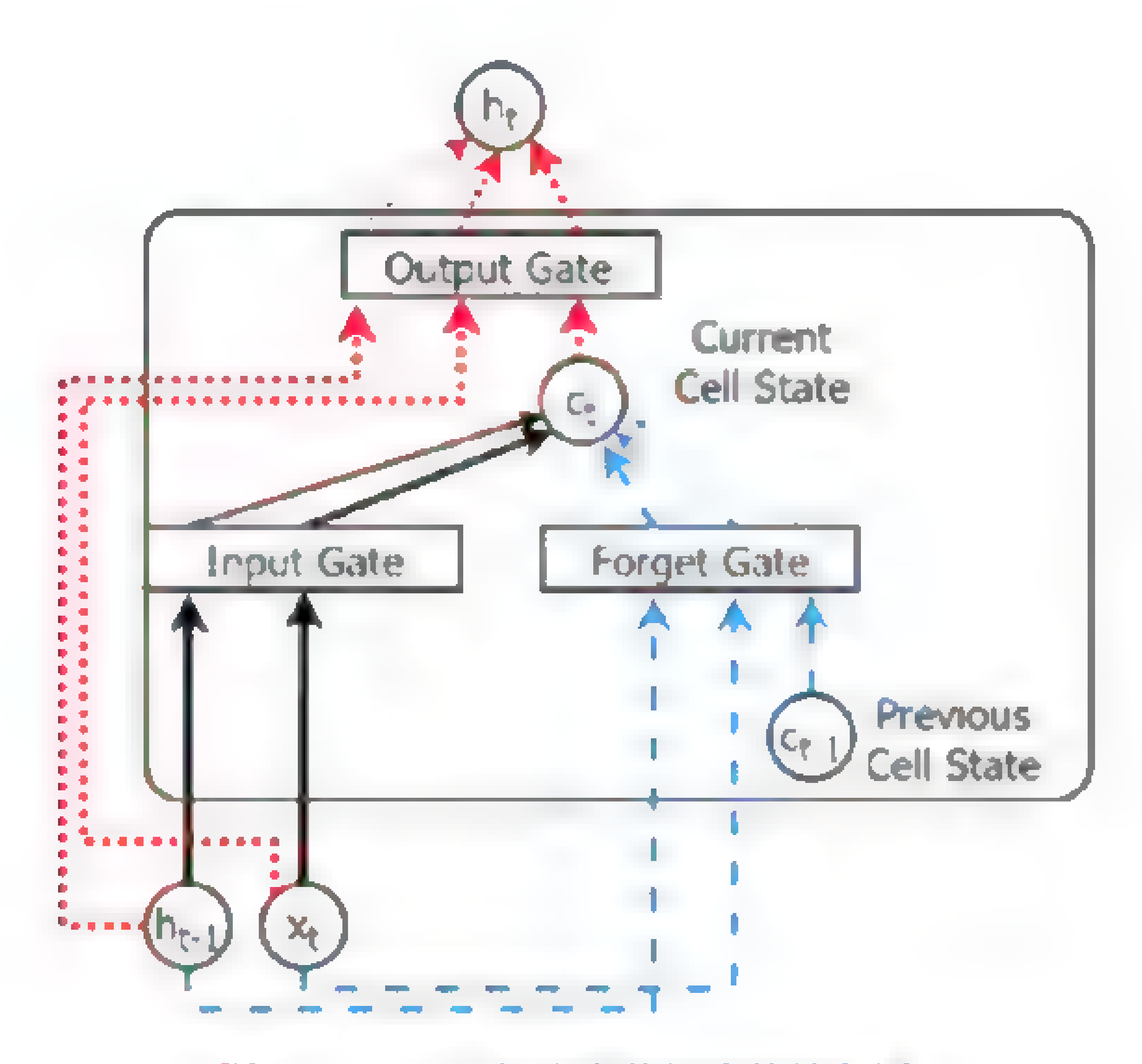


图 7-2 LSTM 细胞中数据流的抽象图

7.2 LSTM 工作原理详解

本节介绍 LSTM 运作的基本原理。由 7.1 节我们知道，LSTM 网络由彼此连接的细胞组成，且每个 LSTM 细胞包含三种类型的门：输入门、遗忘门和输出门，它们分别实现对细胞存储器的写入、复位（或称为重置）和读取功能。这些门不是二进制的，而是模拟的（通常由映射在[0,1]范围内的 Sigmoid 激活函数管理，其中 0 表示总抑制，1 表示总激活）。LSTM 细胞管理两个状态向量：细胞状态和隐藏状态，出于性能原因，它们默认保持独立。当然，在具体操作中也可以通过在创建 BasicLSTMCell 时设置 state_is_tuple=False 来更改此默认行为。图 7-3 提供了具有单个状态单元的 LSTM 细胞块的图示。LSTM 网络的形成与简单的 RNN 是一致的，只是隐藏层中的非线性单元被存储器块替换。此外，与其他 RNN 一样，隐藏层可以附加到任何类型的可微分输出层，具体取决于所需的任务（回归、分类等）。

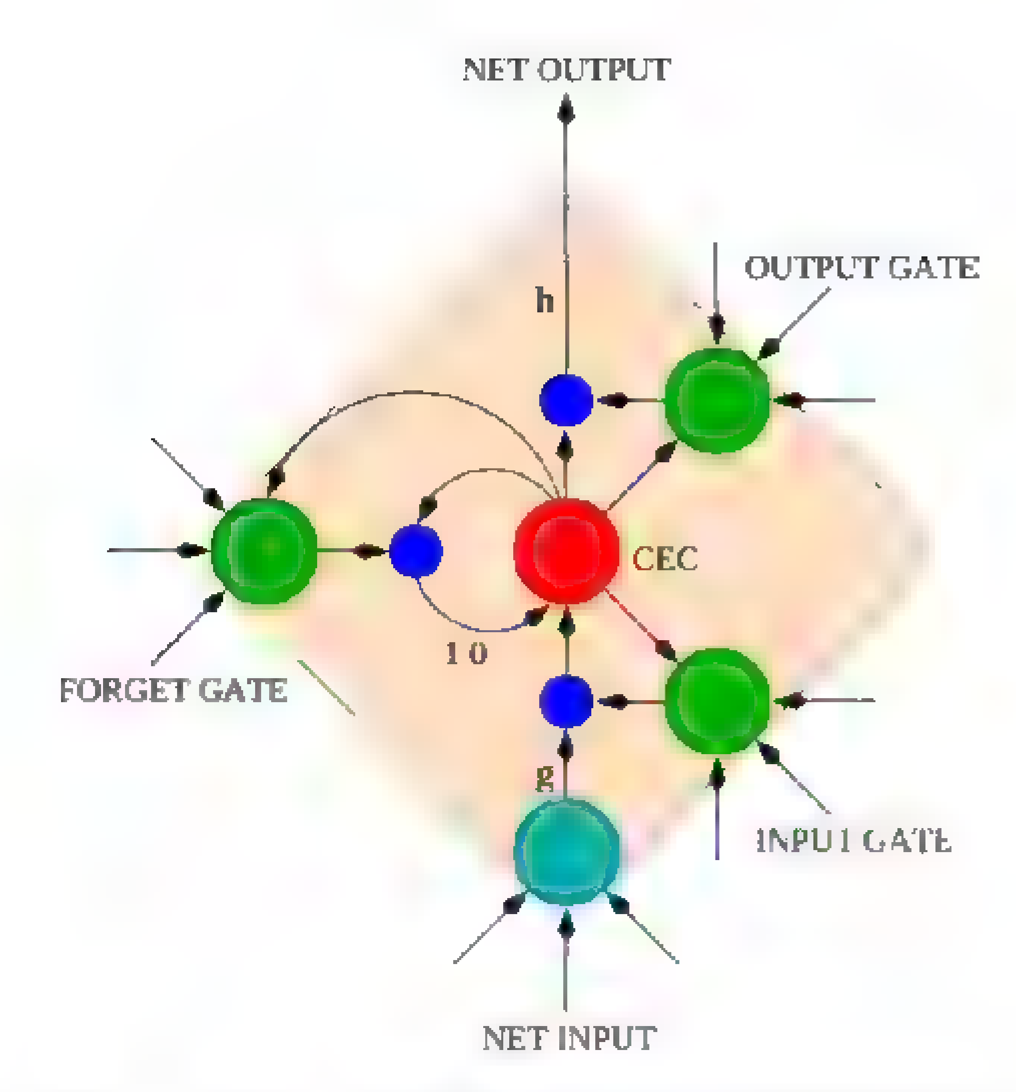


图 7-3 具有一个状态单元的 LSTM 细胞块图示

在图 7-3 中，通过设定权重值为固定（1.0）以重复连接来维持内部细胞状态。三个门从块的内部和外部收集激活情况，并通过乘法单元（小圆圈）控制细胞状态。输入门和输出门控制细胞数据流的输入和输出程度，而遗忘门控制内部细胞状态。细胞输入和输出激活函数（g 和 h）应用于指定位置。实际上，多数情况下该自循环的权重值会视上下文情况而产生变动。

设想一下，日常生活中可拆装的汽车玩具，在组装汽车的过程中，每个时间步骤中都会面临以下状况：

（1）手头有的汽车部件。

（2）回忆或询问旁人汽车组装好后完整的结构图样。

（3）汇集（1）和（2）中的信息，决定将当前的部件组装到哪个位置。

这种生活中遇到的场景与我们的 LSTM 的工作原理其实很相似，比如（1）中你拥有的汽车部件就是当前的外部输入x_t；（2）中其实就是调入之前的记忆/信息；（3）中就是汇集外部输入和x_t之前的记忆来决定输出的结果（y_t）。下面我们通过一个具体的例子对 LSTM 内在逻辑进行解读。

首先，假设这里有一个句子：

```
John gave Mary a puppy.
```

我们输出的内容应该是 John、Mary 和 puppy。这时假设 LSTM 在上面给定的句子后需要输出两个句子：

```
John gave Mary a puppy. ____________________. ____________________.
```

那么后面的两个句子可能是：

```
It barks very loudly. They named it Luna.
```

虽然我们的 LSTM 模型还远未输出诸如此类的实际短语，但是可以学习这些名词和代词之间的关系。例如，It 与 puppy 有关，They 与 John 和 Mary 有关。然后，It 应该学习名词/代词和动词之间的关系。例如，对于 It，动词最后应该有一个 s。对于这些情况，我们结合例句给出了这些单词间的依赖关系，如图 7-4 所示。我们看到短语中的长期记忆（例如 Luna→puppy）和短期记忆（例如 It→barks）的依赖关系，实线箭头代表名词和代词之间的联系，虚线箭头显示了名词/代词和动词之间的联系。

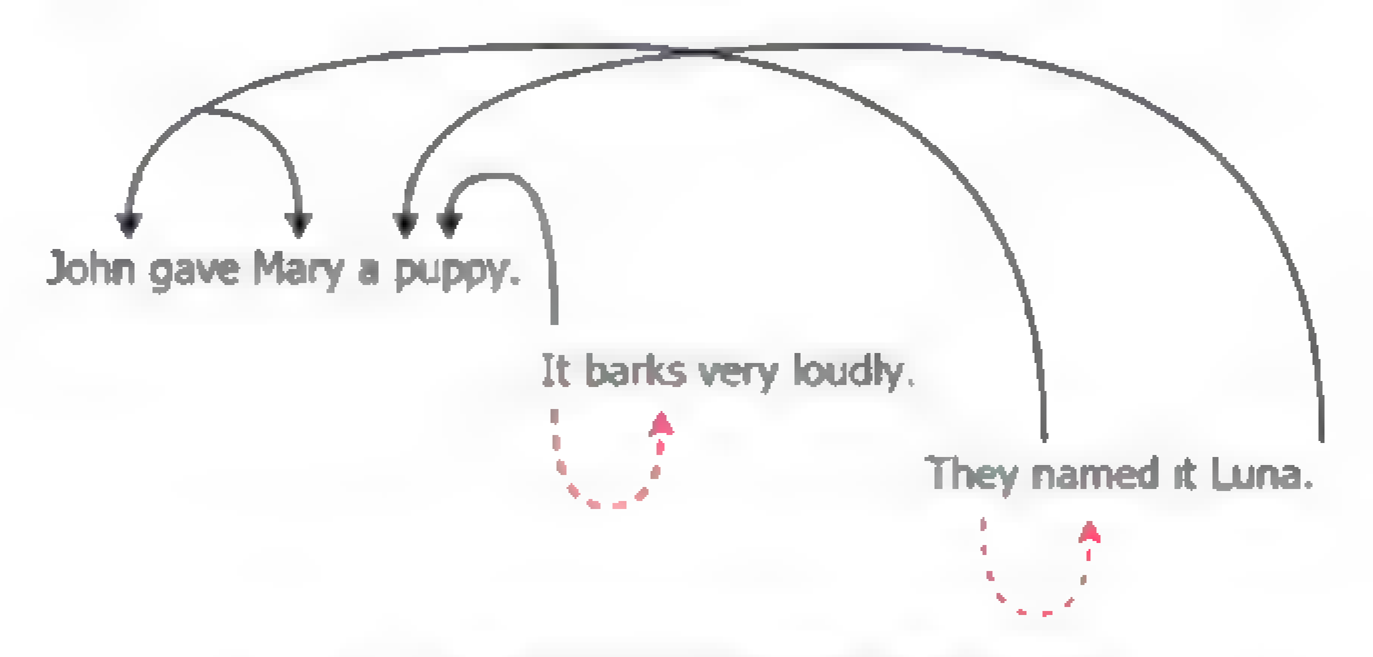


图 7-4 例句中各单词之间的各种关系图

接下来，我们将通过这些内部具体操作来建立单词间的各种关系，以便输出合理的句子文本。

7.2.1 梯度信息如何无损失传递

如何让梯度随着时间的流动不发生指数级消失或者爆炸，这是摆在我们面前的一个重要问题。数学上，无论对 1 做多少次方的运算结果还是 1，如果我们得到的梯度恒为 1，貌似问题就解决了。这时，我们就可以对长时记忆细胞 c 的数学模型进行设计，大概样子如下：

$$c_t = c_{(t-1)}$$

这时，误差反向传播时的导数就是恒定的（即为 1），误差可以一路无损耗地向前传播直到网络起始端。也就是说，我们可以学习到神经网络前端与末端的依赖性。

就像汽车运输一样，运输路线畅通了。接下来我们对数据的装载和卸载进行处理。

7.2.2 将信息装载入长时记忆细胞

由前面得知，输入门i_t将当前输入x_t和前一个最终隐藏状态$h_{(t-1)}$作为输入并计算i_t，具体如下：

$$i_t = \sigma(W_{(ix)}x_t + W_{(ih)}h_{(t-1)} + b_i) \tag{7.1}$$

这里$\sigma(x) = \frac{1}{(1+e^{(-x)})}$。

输入门可以理解为在具有 Sigmoid 激活函数的单隐藏层（标准 RNN 上的隐藏层）处执行的计算。标准 RNN 的隐藏层状态h_t计算公式如下：

$$h_t = \tanh\left(Ux_t + Wh_{(t-1)}\right) \tag{7.2}$$

我们发现除了激活函数不同和增加偏差之外，LSTM 中的输入门i_t和标准 RNN 的隐藏状态h_t在计算上很相似。由上面的内容我们得知，对于输入门i_t计算的结果，如果值为 1，就意味着当前输入的所有信息将被传递到细胞状态；如果值为 0，就意味着当前输入的信息均不能被传递到细胞状态。

接下来，我们要给出由 tanh 层创建的一个候选项向量，记为$\tilde{c}_t$，该向量将会被添加以便稍后能够计算当前细胞状态。

$$\tilde{c}_t = \tanh(W_{(cx)}x_t + W_{(ch)}h_{(t-1)} + b_c) \tag{7.3}$$

至此，涉及的计算都可以在图 7-5 中看到。

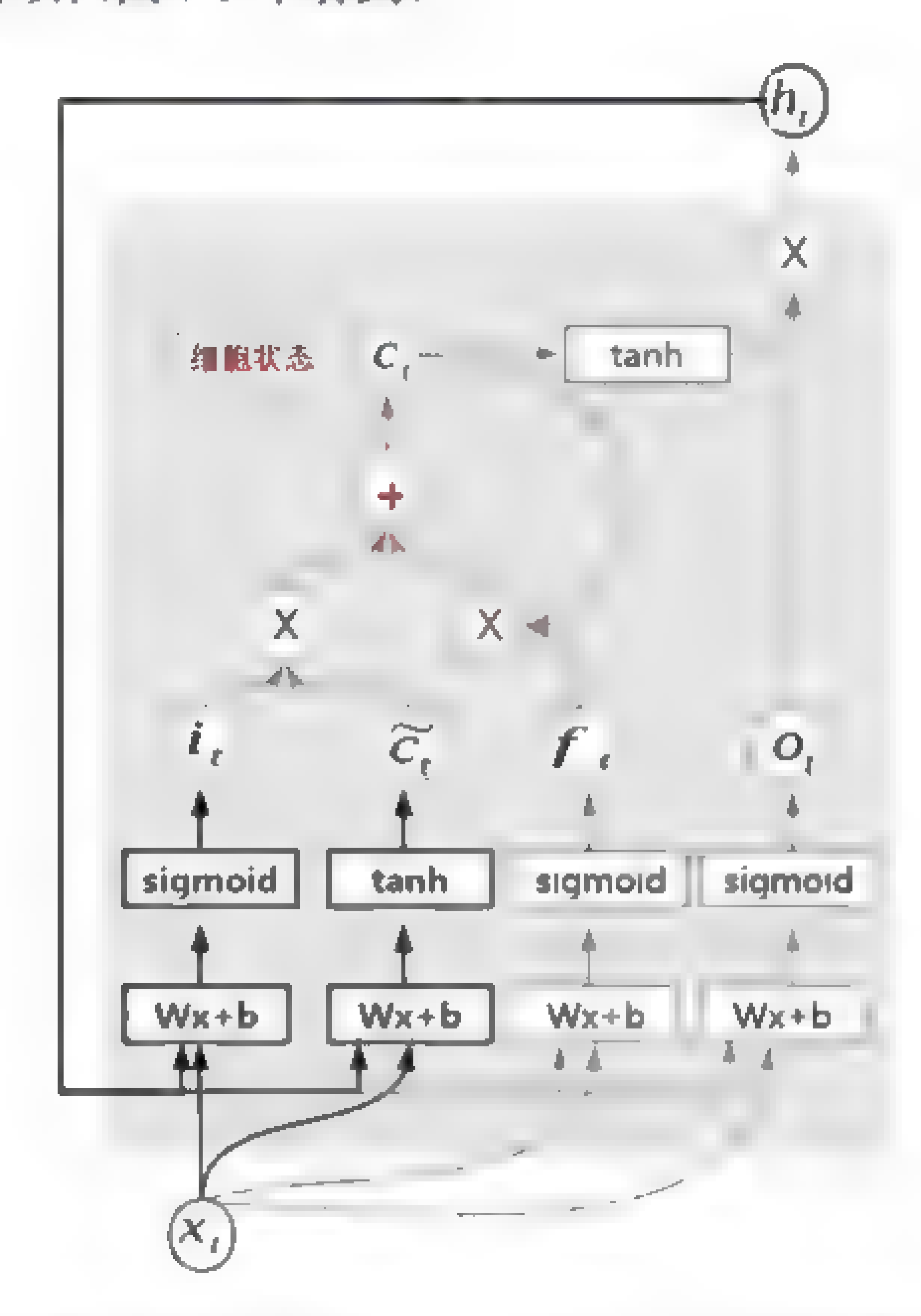


图 7-5　LSTM 中涉及的所有计算（灰色显示），其中的计算i_t和$\tilde{c}_t$以黑色显示

而在示例中，在学习的最初阶段，输入门需要高度激活。LSTM 输出的第一个词就是 it。为此，LSTM 必须知道 puppy 也被称为 it。假设 LSTM 由 5 个神经元来存储状态。我们希望 LSTM 存储 it 所指的是 puppy 信息，也希望 LSTM 学习的另一条信息（在不同的神经元中）是当使用的主语是 it 时，其对应动词的时态应该在词末尾有一个 s。LSTM 需要知道的另一件事是小狗（puppy）可以大声（loudly）吠（barks）。图 7-6 说明了 LSTM 如何在其细胞状态中对这些信息进行编码处理，每个圆圈代表细胞状态的单个神经元（隐藏单元）。

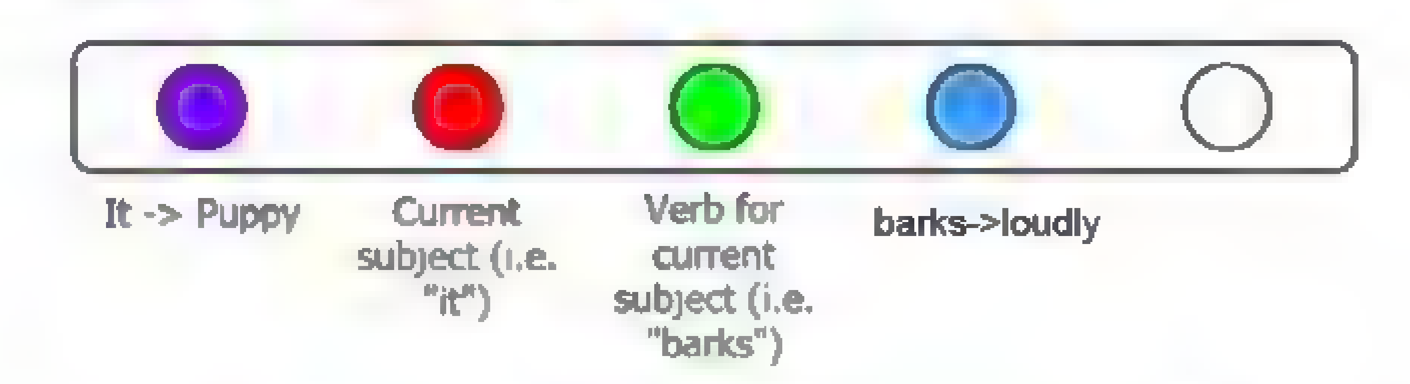


图 7-6　该细胞状态对这些信息进行编码并输出第一个句子的信息

同时我们输出第一个新句子，如下：

```
John gave Mary a puppy. It barks very loudly.
```

7.2.3 更新细胞状态可能产生的问题及解决方案

目前例子中的样本信息量很小，对于输入门影响不大，但现实中样本数据量都很大，如果输入状态处于打开状态，试图记住所有的信息会产生不利的结果：状态 c 的值会变得很大。因为神经网络在输出时需要把 c 激活，而当 c 变得非常大时，像 sigmoid、tanh 这些常见的激活函数的输出就会完全饱和，比如图 7-7 给出的 tanh 激活函数。

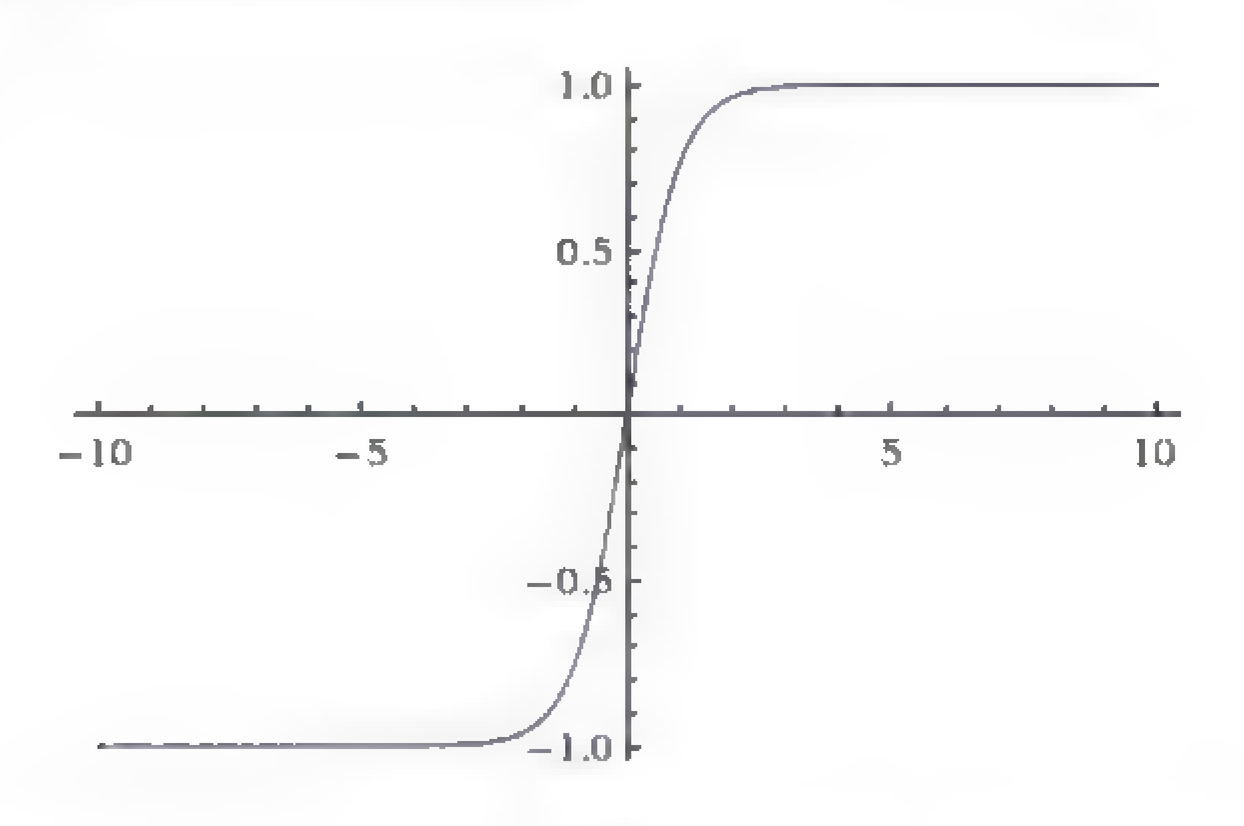


图 7-7　tanh 激活函数

当 c 的容量巨大时，tanh 曲线趋近于 1，这时 c 再增加也没有什么意义，因为已经处于饱和状态，不可能再装载更多信息。我们前面提到的遗忘门就是专门为解决此类问题而设计的，可以派上用场了：每个时刻传递过来的信息，需要通过遗忘门丢弃一些不需要的信息，当然会记住一些需要的信息，这就可以防止上面提到的状态饱和问题。遗忘门的计算公式如下：

$$f_t = \sigma(W_{(fx)}x_t + W_{(fh)}h_{(t-1)} + b_f) \tag{7.4}$$

这里$\sigma(x) = \frac{1}{(1+e^{(-x)})}$。

通过该公式，对于遗忘门而言将会执行如下操作：

- 遗忘门的值为 0 意味着它将不会传递来自$c_{(t-1)}$的信息来更新c_t。
- 遗忘门的值为 1 意味着它将传递来自$c_{(t-1)}$的所有信息来更新c_t。

接下来，我们将看到遗忘门如何帮助我们预测下一句：

```
They named it Luna.
```

这时可以看到，我们正在寻找的新关系是约翰（John）和玛丽（Mary）以及他们（They）之间的关系。因此，我们不再需要有关主语它（It）的信息以及动词 barks 的行为，因为主语是约翰（John）和玛丽（Mary）。我们通过遗忘门用当前主语（They）和其对应动词（named）来替换存

储在当前神经元中的主语和对应的动词信息，具体如图 7-8 所示。

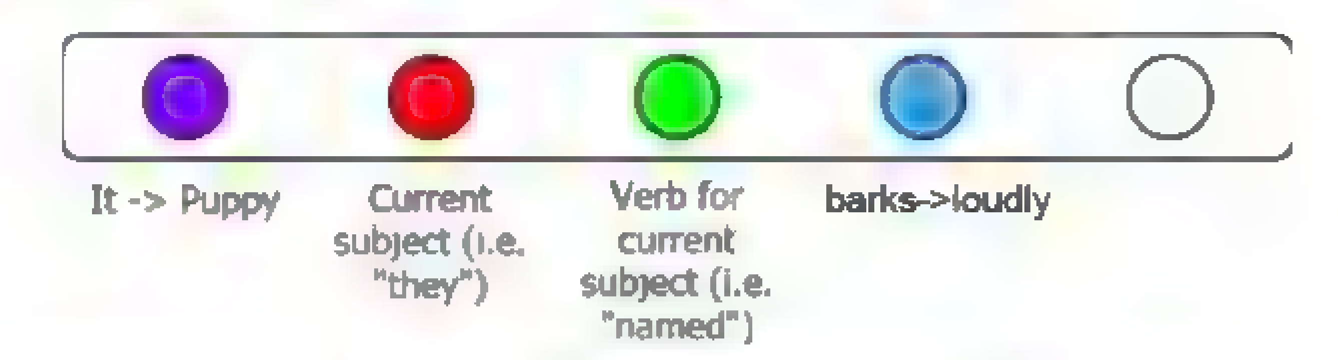


图 7-8 左边起第二和第三个神经元的信息（It →barks）被新信息（They →named）所取代

就涉及输入门和遗忘门的权重值而言，图 7-9 中说明了这些转变过程。我们不会改变维持 It→puppy 这种关系的神经元的状态，因为 puppy 在最后一句中显示为一个被操作对象。这是通过设置将 It → puppy 从$c_{(t-1)}$连接到c_t的权重值设置为 1 来完成的。然后我们将用新的主语和动词来更新维持当前主语和当前动词信息的神经元信息。这是通过将该神经元的遗忘权重值f_t设置为 0 来实现的。接着，我们将当前主语和动词连接到相应神经元状态的权重值i_t设置为 1，我们可以将$\tilde{c}_t$视为哪些新信息（例如来自当前输入x_t的新信息）应该被带进细胞状态的实体中。

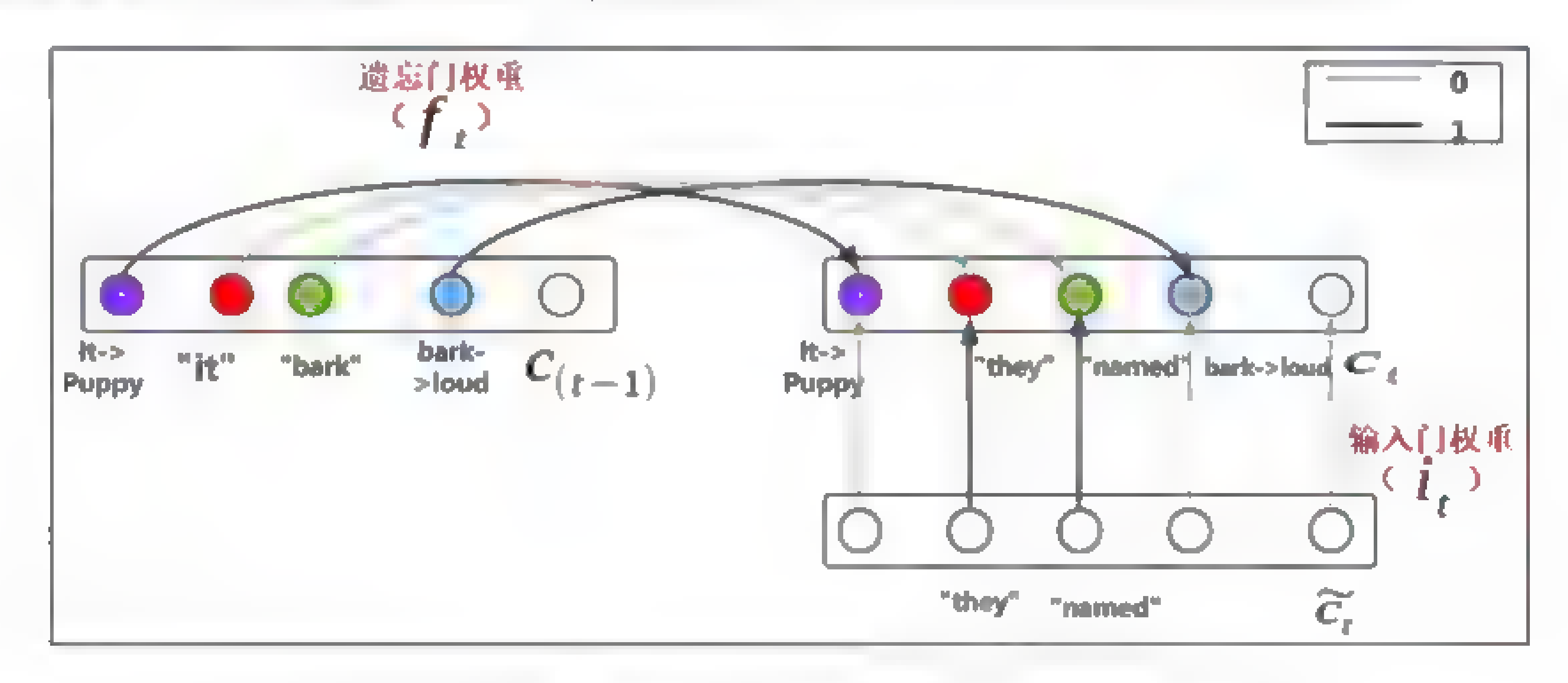


图 7-9 用先前状态$c_{(t-1)}$和候选项$\tilde{c}_t$来计算细胞状态c_t

当前细胞状态将更新如下：

$$c_t = f_t c_{(t-1)} + i_t \tilde{c}_t \tag{7.5}$$

换句话说，当前状态是以下的组合：

- 前一个细胞状态遗忘/记住哪些信息。
- 要添加/丢弃当前输入的信息。

图 7-10 显示了当前在 LSTM 中进行的所有计算的内容。

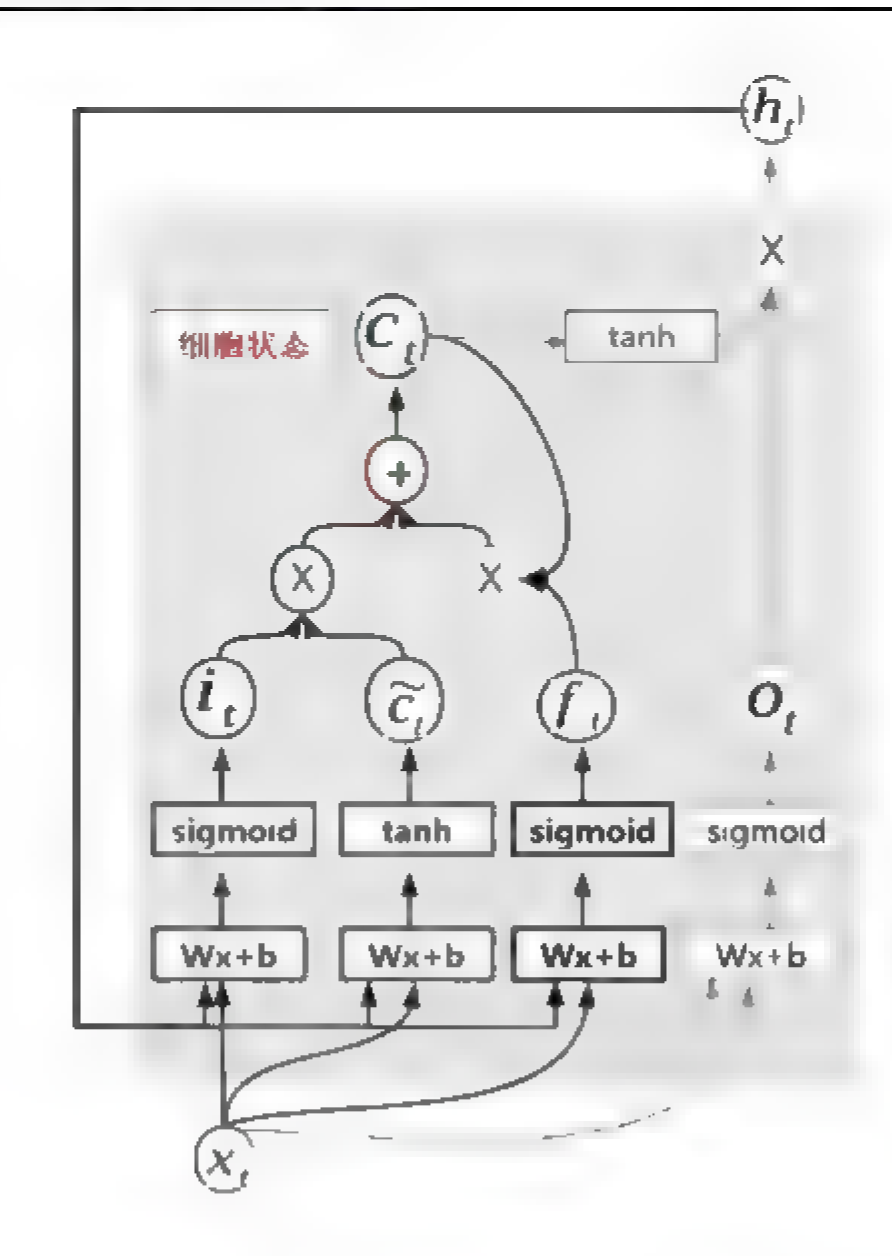


图 7-10 当前在进行的所有计算的内容（$i_t, f_t, \tilde{c}_t, c_t$）

在学习之后，完整的细胞状态如图 7-11 所示。

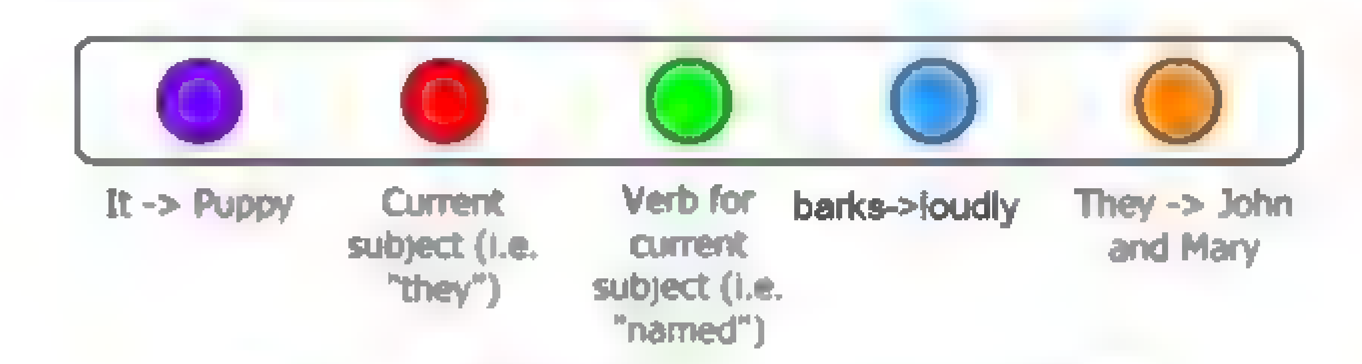


图 7-11 输出两个句子后，完整的细胞状态图

接下来，我们将看看如何计算 LSTM 细胞的最终状态（h_t）：

$$o_t = \sigma\left(W_{(ox)}x_t + W_{(oh)}h_{(t-1)} + b_o\right) \tag{7.6}$$

$$h_t = o_t \tanh(c_t) \tag{7.7}$$

在示例中，需要输出以下句子：

```
They named it Luna.
```

为此，我们不需要倒数第二个神经元来计算这个句子，因为它包含有关小狗（puppy）如何吠（barks）的信息，而这句话是关于小狗（puppy）名字的。这么看来，在最后一句的预测中就可以忽略最后一个神经元（含有 barks->loudly 的关系）。这正是o_t所做的，它将忽略不必要的记忆信息，而仅在计算 LSTM 细胞的最终输出时从细胞状态检索相关的记忆信息。图 7-12 给出了 LSTM 细胞内部的逻辑。

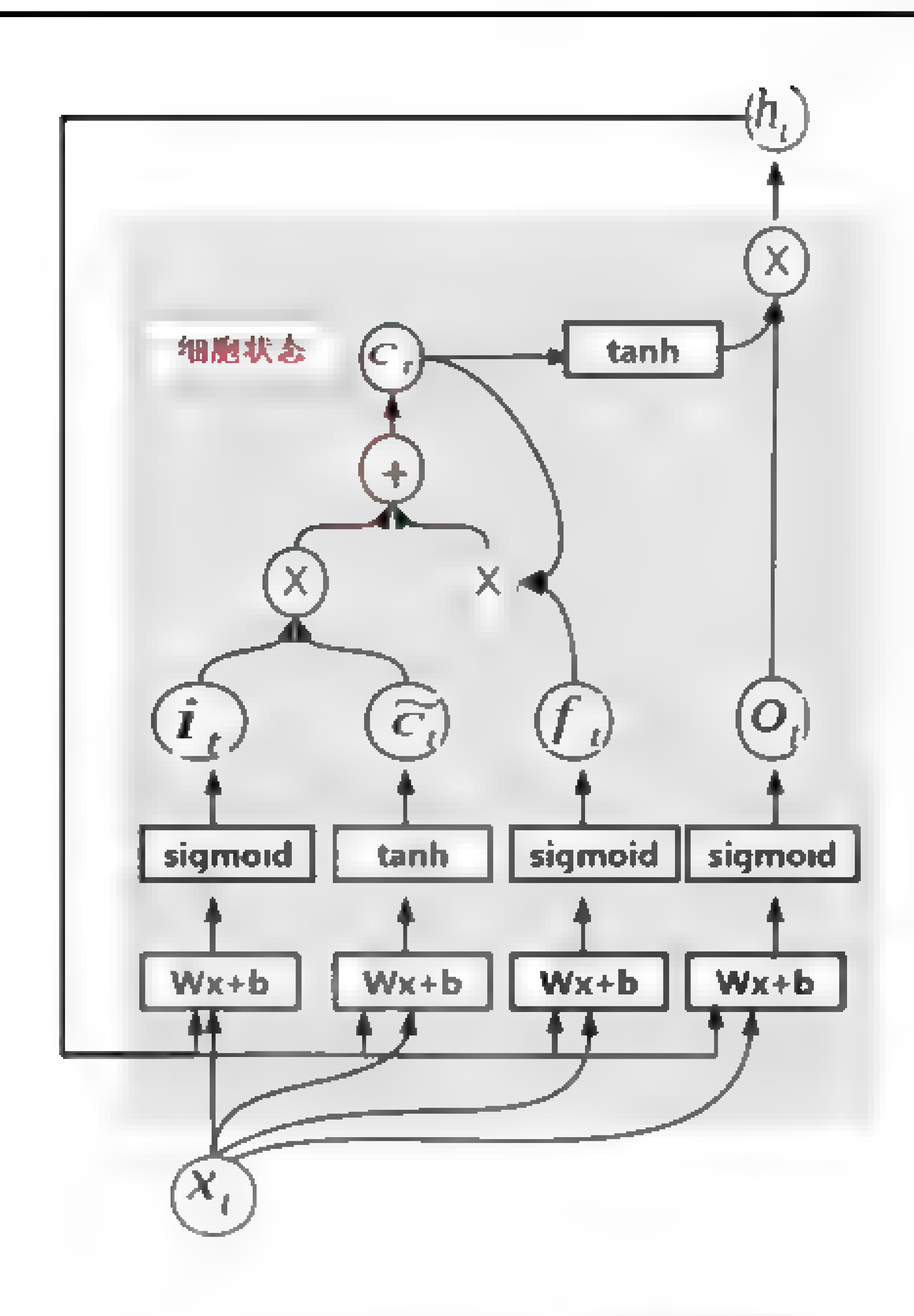


图 7-12　完整的 LSTM 细胞的内部逻辑图

在这里，我们总结一下在 LSTM 细胞内部发生操作的所有相关方程式。

$$i_t = \sigma(W_{(ix)}x_t + W_{(ih)}h_{(t-1)} + b_i)$$

$$\tilde{c_t} = \tanh(W_{(cx)}x_t + W_{(ch)}h_{(t-1)} + b_c)$$

$$f_t = \sigma(W_{(fx)}x_t + W_{(fh)}h_{(t-1)} + b_f)$$

$$c_t = f_t c_{(t-1)} + i_t\tilde{c_t}$$

$$o_t = \sigma(W_{(ox)}x_t + W_{(oh)}h_{(t-1)} + b_o)$$

$$h_t = o_t\tanh(c_t)$$

7.2.4　LSTM 模型输出

现在从更宽的角度来看一下关于学习序列样本数据的问题。将 LSTM 细胞随时间变化展开，以便展示这些细胞是如何连接在一起的，从而挖掘出这些细胞接收先前的状态后去计算下一个细胞状态，如图 7-13 所示。

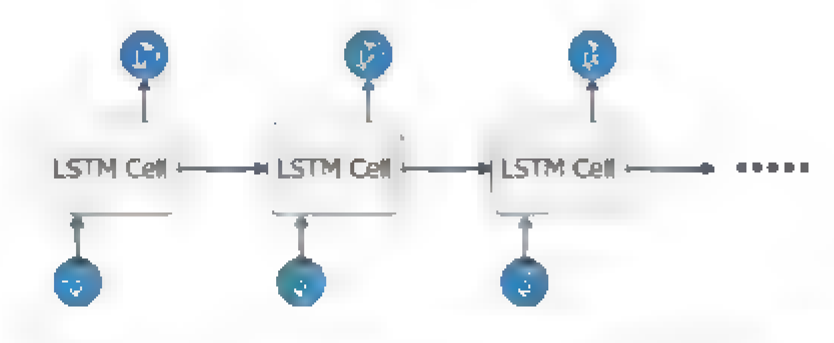


图 7-13　LSTM 细胞随时间变化的连接图

但是实际上，这些还不足以做一些有用的工作。正如我们所看到的那样，即使可以创建一个真正能够对序列数据进行建模的 LSTM 链，我们仍然无法进行输出或预测，而倘若想要使用 LSTM 实际学到的信息，我们需要一种能够从 LSTM 细胞中抽取最终输出信息的方法。因此，我们将在 LSTM 顶部固定一个 softmax 层（包含权重值W_s和偏差b_s），最终的输出可以由以下等式获得：

$$y_t = softmax(W_s h_t + b_s) \tag{7.8}$$

现在，带有 softmax 层的 LSTM 的最终图像如图 7-14 所示。

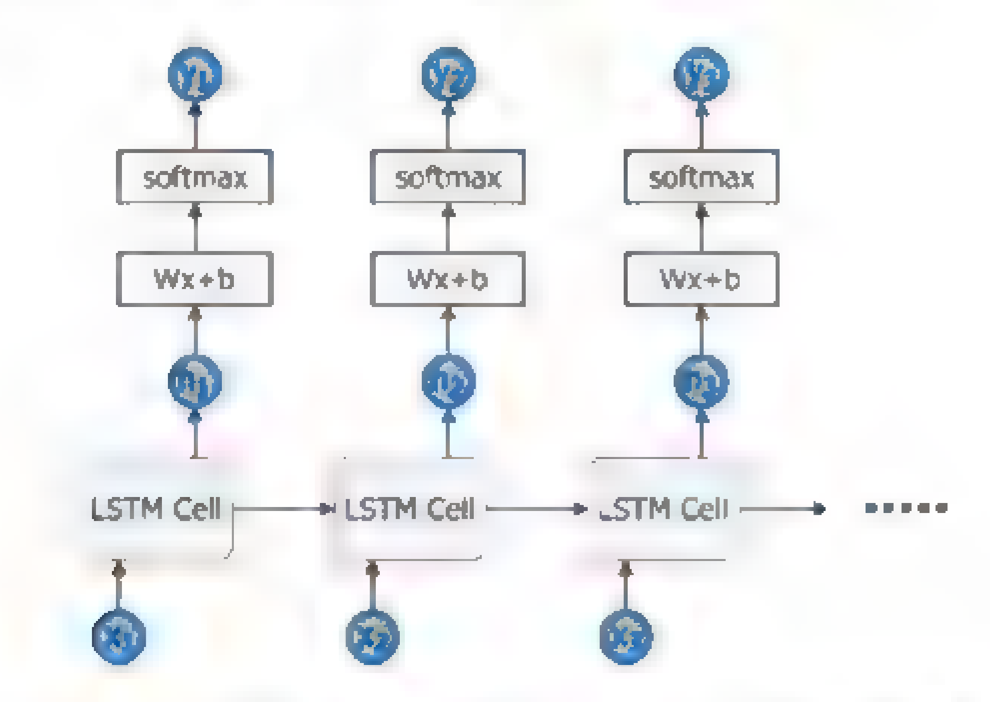


图 7-14　带有 softmax 层的 LSTM 随时间变化图

7.3　LSTM 与标准 RNN 的区别

本节将要对 LSTM 与标准 RNN 进行比较分析。实际上，与标准 RNN 相比，LSTM 具有更复杂的结构。主要区别之一是 LSTM 具有两种不同的状态：细胞状态 c_t 和最终隐藏状态，而 RNN 仅具有单个隐藏状态。另一个主要区别在于，由于 LSTM 具有三个不同的门（Gate），因此当计算最终隐藏状态时，LSTM 对如何处理当前输入和先前的细胞状态具有更多的控制空间。

LSTM 自带的这两种不同的状态是非常具有竞争力的。起初，LSTM 模型的核心贡献（Hochreiter 和 Schmidhuber，1997）是引入自循环的绝妙想法，以便产生梯度长时间不间断流动的路线。这里的一个关键性突破就是自循环的权重值视上下文情况而定，而不是固定的（Gerset 等，2000）。门控这个自循环（由另一个隐藏单元控制）的权重值累积的时间尺度可以产生动态变化，此时即便 LSTM 的参数是固定的，累积的时间尺度也能够因输入序列而发生变化，这是由于时间常数是模型本身的输出。由此可见，LSTM 这种机制的存在，即使细胞状态发生快速变化，最终隐藏状态仍将变动得更缓慢。因此，虽然细胞状态正在学习短期和长期依赖性，但最终隐藏状态可以仅反映短期依赖性或仅反映长期依赖性或两者都反映。

由此可见，LSTM 是一个原则性更强的方法（特别是与标准 RNN 相比），因为它可以更加自信地控制当前输入和之前细胞状态对当前细胞状态的贡献程度。此外，输出门可以更好地控制细胞状态对最终隐藏状态的贡献程度。在图 7-15 中，我们给出了标准 RNN 和 LSTM 的比较示意图，显示出两个模型之间的差异。

综上所述，通过设计和维持两种不同的状态，LSTM 可以学习短期和长期依赖关系，这有助于

解决梯度消失的问题，我们将在下一节中讨论。

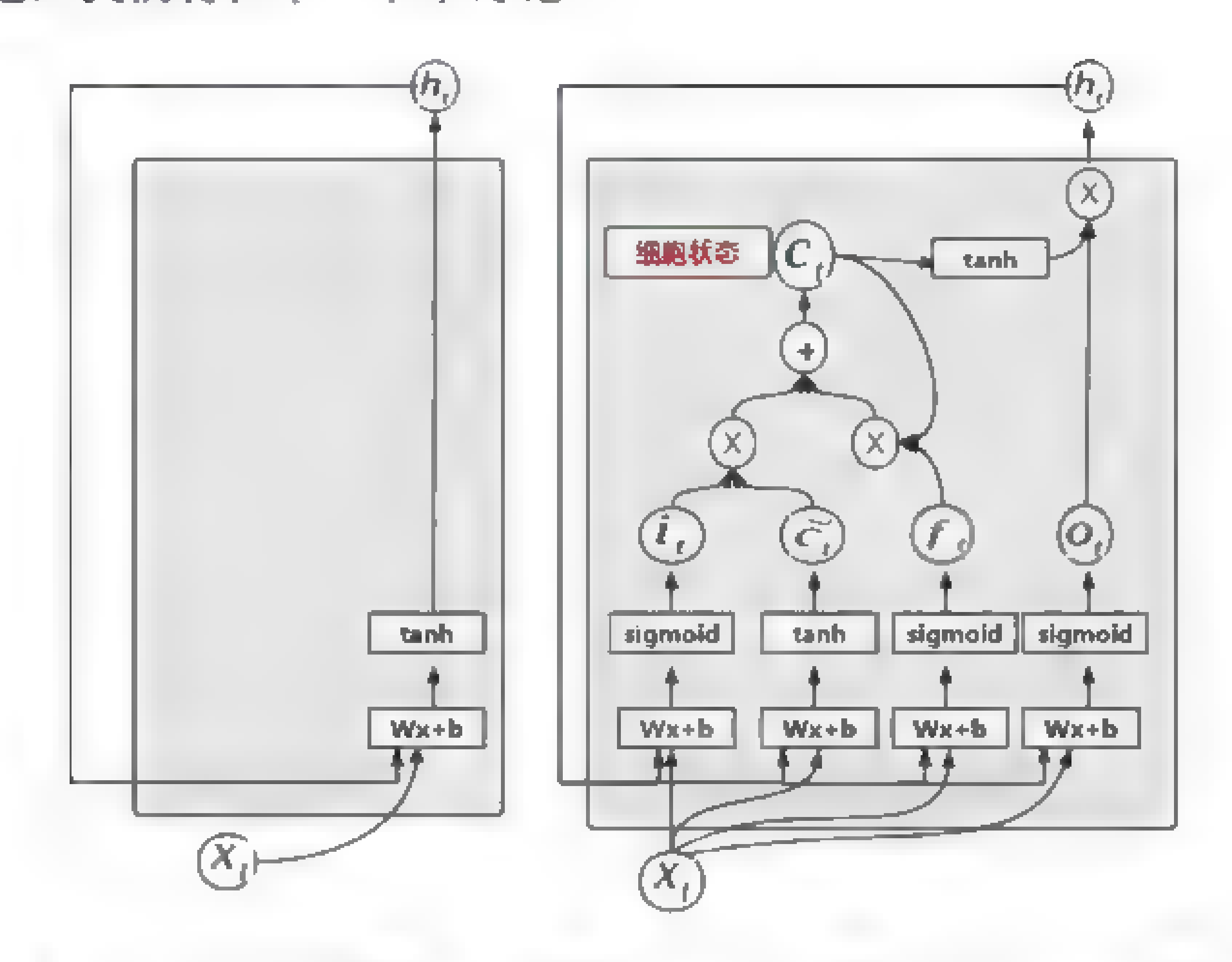


图 7-15　二者对比图示：左侧为 RNN，右侧为一个 LSTM 细胞

7.4　LSTM 如何避免梯度消失和梯度爆炸问题

正如我们前面所讨论的，即使 RNN 在理论上是合理的，但在实践中它们也存在严重的缺陷。也就是说，当使用通过时间反向传播（BPTT）时，梯度会迅速减小，这使我们只能传播几个时间步长的信息。这样一来，我们只能存储极少时间步长的信息，进而只能拥有短期记忆。这反过来限制了 RNN 在实际序列任务中的应用性。

通常有用且有趣的序列工作任务（例如股票市场预测或语言建模）需要具有学习和存储长期依赖关系的能力。以下示例用于预测下一个单词：

```
John is a talented student. He is an A-grade student and plays rugby and cricket.
All the other students envy ______.
```

对我们来说，这是一项非常容易的任务。答案是约翰（John）。但是，对于 RNN 来说，这是一项艰巨的任务。我们试图预测一个答案，该答案位于文本的开头。此外，为了解决这个任务，我们需要一种在 RNN 状态下存储长期依赖关系的方法。这正是 LSTM 旨在解决的任务类型。

在第 6 章的循环神经网络中，我们讨论了如何在没有任何非线性函数的情况下出现梯度消失/爆炸。我们现在将看到，即使存在非线性项，它仍然可能发生。为此，我们将看到导数项$\frac{\partial h_t}{\partial h_{(t-k)}}$如何用于标准 RNN 和 LSTM（LSTM 的是$\frac{\partial c_t}{\partial c_{(t-k)}}$）网络。正如我们在第 6 章中所学到的，这是导致梯度消失的关键项。

假设标准 RNN 的隐藏状态计算如下：

$$h_t = \sigma(W_x x_t + W_h h_{(t-1)})$$

为了简化计算，我们可以忽略当前与输入相关的项，并将重点放在循环部分，这将给出以下等

式：

$$h_t = \sigma(W_t h_{(t-1)})$$

如果我们计算前面的$\frac{\partial h_t}{\partial h_{(t-k)}}$项，将得到以下结果：

$$\frac{\partial h_t}{\partial h_{(t-k)}} = \prod_{i=0}^{k-1} W_h \sigma(W_h h_{(t-k+i)})(1-\sigma(W_h h_{(t-k+i)}))$$

$$=W_t^k \prod_{i=0}^{k-1} \sigma\left(W_h h_{(t-k+i)}\right)\left(1-\sigma\left(W_h h_{(t-k+i)}\right)\right)$$

现在让我们看看当$W_h h_{(t-k+i)}$<< 0 或$W_h h_{(t-k+i)}$>> 0 时会发生什么。在这两种情况下，$\frac{\partial h_t}{\partial h_{(t-k)}}$将开始接近 0，从而发生梯度消失。当$W_h h_{(t-k+i)}$= 0 时，对于 Sigmoid 激活函数而言，梯度是最大的（0.25），当乘以多个时间步长时，整体梯度变得非常小。此外，W_h^k项（可能由于初始化不合理）也会导致梯度爆炸或消失。然而，与由$W_h h_{(t-k+i)}$<< 0 或$W_h h_{(t-k+i)}$>> 0 而导致的梯度消失相比，由W_h^k项引起的梯度消失/爆炸相对容易解决（认真合理初始化权重值和进行梯度裁剪）。

现在让我们看一下细胞状态，由下式给出：

$$c_t = f_t c_{(t-1)} + i_t \tilde{c}_t$$

这是 LSTM 中发生的所有遗忘门应用的产物。但是，如果以类似的方式为 LSTM 计算$\frac{\partial c_t}{\partial c_{(t-k)}}$（忽略$W_{fx}x_t$项和$b_f$，因为它们是非循环的），将得到以下结果：

$$\frac{\partial c_t}{\partial c_{(t-k)}} = \prod_{i=0}^{k-1} \sigma(W_{fh} h_{(t-k+i)})$$

在这种情况下，如果$W_h h_{(t-k+i)}$<< 0，那么梯度将消失；另一方面，如果$W_h h_{(t-k+i)}$>> 0，那么导数将比标准 RNN 中的变动速度慢得多。此外，当使用压缩函数时，由于$\frac{\partial c_t}{\partial c_{(t-k)}}$很大（梯度爆炸期间可能发生的事情），梯度不会爆炸。此外，当$W_h h_{(t-k+i)}$>> 0 时，我们得到接近 1 的最大梯度，这意味着梯度不会像我们在 RNN 中看到的那样迅速减小（当梯度达到最大值时）。最后，在推导中没有诸如W_h^k之类的项。然而，对于$\frac{\partial h_t}{\partial h_{(t-k)}}$的推导更为棘手。让我们看看这些项是否存在于$\frac{\partial h_t}{\partial h_{(t-k)}}$的推导中。如果我们计算出它的这些推导，将得到以下的式子：

$$\frac{\partial h_t}{\partial h_{(t-k)}} = \frac{\partial(o_t \tanh(c_t))}{\partial h_{(t-k)}}$$

一旦解决了这个问题，我们就会得到这样的形式：

$$\tanh(.)\sigma(.)[1-\sigma(.)]w_{oh} + \sigma(.)[1-\tanh^2(.)](c_{(t-1)}\sigma(.)[1-\sigma(.)]w_{fh} + \sigma(.)[1-\tanh(.)^2]w_{ch} + \tanh(.)\sigma(.)[1-\sigma(.)]w_{ih})$$

我们不关心 σ (.)或 tanh (.)内的内容，因为无论是什么值，它都将在（0,1）或（-1,1）的范围之内。如果我们用一些常见的符号（比如 γ (.)）来替换 σ (.)、[1-σ (.)] 、tanh (.)和[1-tanh (.)^2]

项，将得到如下式子：

$$\gamma(.)w_{oh} + \gamma(.)[c_{(t-1)}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}]$$

或者得到以下结果（假设外部 γ(.)被方括号内的每个 γ(.)项吸收）：

$$\gamma(.)w_{oh} + c_{(t-1)}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$$

这将给出以下内容：

$$\frac{\partial h_t}{\partial h_{(t-k)}} \approx \prod_{i=0}^{k-1} \gamma(.)w_{oh} + c_{(t-1)}\gamma(.)w_{fh} + \gamma(.)w_{ch} + \gamma(.)w_{ih}$$

这意味着虽然项$\frac{\partial c_t}{\partial c_{(t-k)}}$对任何$W_h^k$项是安全的，但是对$\frac{\partial h_t}{\partial h_{(t-k)}}$不是。因此，在初始化 LSTM 的权重值时必须谨慎，我们也应该使用梯度裁剪。

注意

尽管如此，梯度消失的稳定性对于 LSTM 并不像 RNN 那样重要。因为c_t仍然可以存储长期依赖性而不受梯度消失的影响，并且如果需要，h_t可以从c_t处检索长期依赖性。

7.5　优化 LSTM

正如我们在学习 RNN 时所看到的那样，拥有坚实的理论基础并不一定能保证它们在实践中表现最佳。这是由于计算机数值精度的限制所致，LSTM 也是如此。拥有复杂的设计，允许更好地建模数据中的长期依赖性，这本身并不意味着 LSTM 将输出完全准确的预测。因此，已经开发了许多扩展来帮助 LSTM 在预测阶段表现得更好。在这里，我们将讨论几个这样的改进：贪婪采样、集束搜索、使用词向量而不是词的 One-Hot 编码表示以及使用双向 LSTM。

7.5.1　贪婪采样

如果我们始终试图以最高概率预测词，LSTM 将倾向于产生非常单一的结果。例如，它会在切换到另一个单词之前多次重复单词 the。

解决这个问题的一种方法是使用贪婪采样（Greedy Sampling），我们从中选择预测最佳的 n 和样本。这有助于打破预测的单一性。

让我们考虑前一个例子的第一句话：

```
John gave Mary a puppy.
```

我们从第一个单词开始，并想要预测接下来的 4 个单词：

```
John ____ ____ ____ ____.
```

如果我们尝试确定性地选择样本，LSTM 可能倾向于输出如下内容：

```
John gave Mary gave John.
```

但是，通过从词汇表里的一个单词子集中采样下一个单词，LSTM 被迫改变预测并可能输出以下内容：

```
John gave Mary a puppy.
```

或者提供以下输出：

```
John gave puppy a puppy.
```

但是，即使贪婪采样有助于为生成的文本添加更多变化，此方法也不能保证输出始终是真实的，尤其是在输出较长的文本序列时。接下来我们看一种更好的搜索技术，它实际上在预测之前的几个步骤中是超前运行的。

7.5.2 集束搜索

集束搜索（Beam Search）是一种帮助 LSTM 改善预测质量的方法。在此，通过解决搜索问题找到预测。集束搜索的关键思想是一次产生（b+1）个输出（$y_t, y_{(t+1)}, y_{(t+2)}, \ldots, y_{(t+b)}$）而不是单个输出$y_t$。这里，b 被称为集束的长度，并且产生的 b 个输出被称为集束。从技术上讲，我们选择具有最高联合概率$p(y_t, y_{(t+1)}, y_{(t+2)}, \ldots, y_{(t+b)} \mid x_t)$的集束，而不是选择单一项的最高概率$p(y_t \mid x_t)$。在进行预测之前，我们将会进一步研究未来，这通常会带来更好的结果。

让我们通过前面的例子来理解集束搜索：

```
John gave Mary a puppy.
```

这里，我们将逐个进行单词预测。最初我们有以下内容：

```
John ____ ____ _ _____.
```

假设 LSTM 使用集束搜索产生例句，那么每个单词的概率可能就像图 7-16 中的那样。假设集束长度 b = 2，我们将在搜索的每个阶段考虑 n = 3 个最佳候选项。搜索树如图 7-16 所示。



图 7-16 集束搜索的空间结构，这里 n=3 且 b=2

我们从单词 John 开始，得到词汇表中所有单词的概率。在例子中，当 n = 2 时，我们为树的下一级选择最好的三个候选项：gave、Mary 和 puppy。这里特别说明，这些可能不是 LSTM 实际找到的候选项，仅用作示例而已。然后从这些选定的候选项中，树的下一级继续延伸。在那里，搜索将重复，直至搜索到树中的深度为 b，我们将挑选最好的三个候选项。

图 7-16 中给出了最高联合概率的路径（P（gave , Mary | John） 0.09），这里用粗线箭头突出显示。这是一种更好的预测机制，因为它会为诸如 John gave Mary 之类的短语返回更高的概率或奖励，而不是 John Mary John 或 John gave。

注意，贪婪采样和集束搜索产生的输出在我们的示例中是相同的，这是一个包含 5 个单词的简单句子。然而，当我们将其处理以输出一篇小文章时，情况并非如此。事实上，通过集束搜索产生的结果将比通过贪婪采样产生的结果更加真实，语法更加正确。

7.5.3 使用词向量

另一种优化 LSTM 性能的流行方法是使用词向量而不是使用 One-Hot 编码向量作为 LSTM 的输入。让我们结合上面的例子来进行探讨。假设我们想要从一些随机单词中生成文本，它将是以下内容：

```
John ____ ____ _ _____.
```

我们根据以下句子训练了 LSTM 模型：

```
John gave Mary a puppy. Mary has sent Bob a kitten.
```

假设将词向量定位如图 7-17 所示。

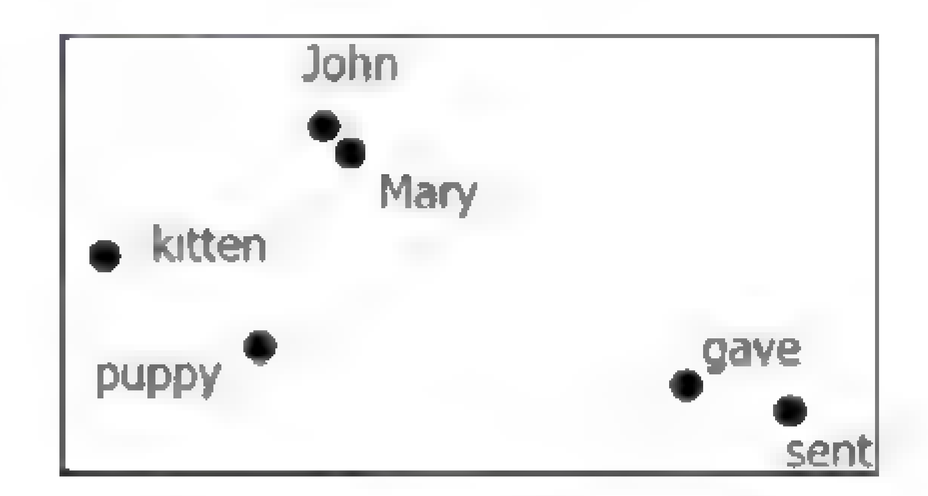


图 7-17　二维空间中的词向量拓扑

这些词向量的数字形式可能如下：

```
kitten: [0.5, 0.3, 0.2]
puppy: [0.49, 0.31, 0.25]
gave: [0.1, 0.8, 0.9]
```

可以看出（kitten , puppy）的距离小于（kitten , gave）的距离。但是，如果我们使用 One-Hot 编码，它们将如下：

```
kitten: [ 1, 0, 0, ...]
puppy: [0, 1, 0, ...]
gave: [0, 0, 1, …]
```

而此时，（kitten , puppy）的距离和（kitten , gave）的距离相等。正如所看到的那样，One-Hot 编码向量不能捕获单词之间的正确关系，且看到所有单词彼此之间的距离都相等。然而，词向量能

够捕获这种关系，且这种关系更适合作为 LSTM 的特征。

由此可见，使用词向量，LSTM 将学习到更好的词之间的关系。例如，使用词向量，LSTM 将学习以下内容：

```
John gave Mary a kitten.
```

这非常接近以下内容：

```
John gave Mary a puppy.
```

此外，它与以下内容完全不同：

```
John gave Mary a gave.
```

但是，如果使用 One-Hot 编码向量，就不会出现这种情况。

7.5.4 双向 LSTM

使双向 LSTM（Bidirectional LSTM，BiLSTM）是提高 LSTM 预测质量的一种方法。利用该方法可以训练 LSTM 从开始到末端和从末端到开始读取数据。在训练 LSTM 期间，将创建如下数据集。

考虑以下两句话：

```
John gave Mary a _____. It barks very loudly.
```

在这个阶段，我们希望 LSTM 能够合理地填补第一个句子中的缺失数据。

如果从头开始读到缺失的单词，那么将是：

```
John gave Mary a _____.
```

这没有提供关于缺失单词上下文的足够信息以便正确地填充单词。但是，如果同时双向读取，那么将是以下内容：

```
John gave Mary a _____.
_____. It barks very loudly.
```

如果我们使用这两个部分创建数据，就足以预测丢失的单词应该像 dog 或 puppy。因此，从两个方向读取数据可以显著优化某些问题的解决思路。此外，这增加了神经网络可用的数据量并提高了其性能。

BiLSTM 的另一个应用是神经网络机器翻译，将源语言的句子翻译成目标语言的句子。由于一种语言的翻译与另一种语言之间没有特定的一致性，因此了解源语言的过去和未来有助于更好地理解上下文，从而产生更好的翻译效果。例如，将菲律宾语翻译成英语的翻译任务。在菲律宾语中，句子通常按顺序写成 verb-object-subject，而在英语中，则是 subject-verb-object。在这个翻译任务中，向前和向后阅读句子以进行良好的翻译。

BiLSTM 本质上是两个独立的 LSTM 网络。一个网络从头到尾学习数据，另一个网络从尾到头学习数据。在图 7-18 中说明了 BiLSTM 网络的架构。

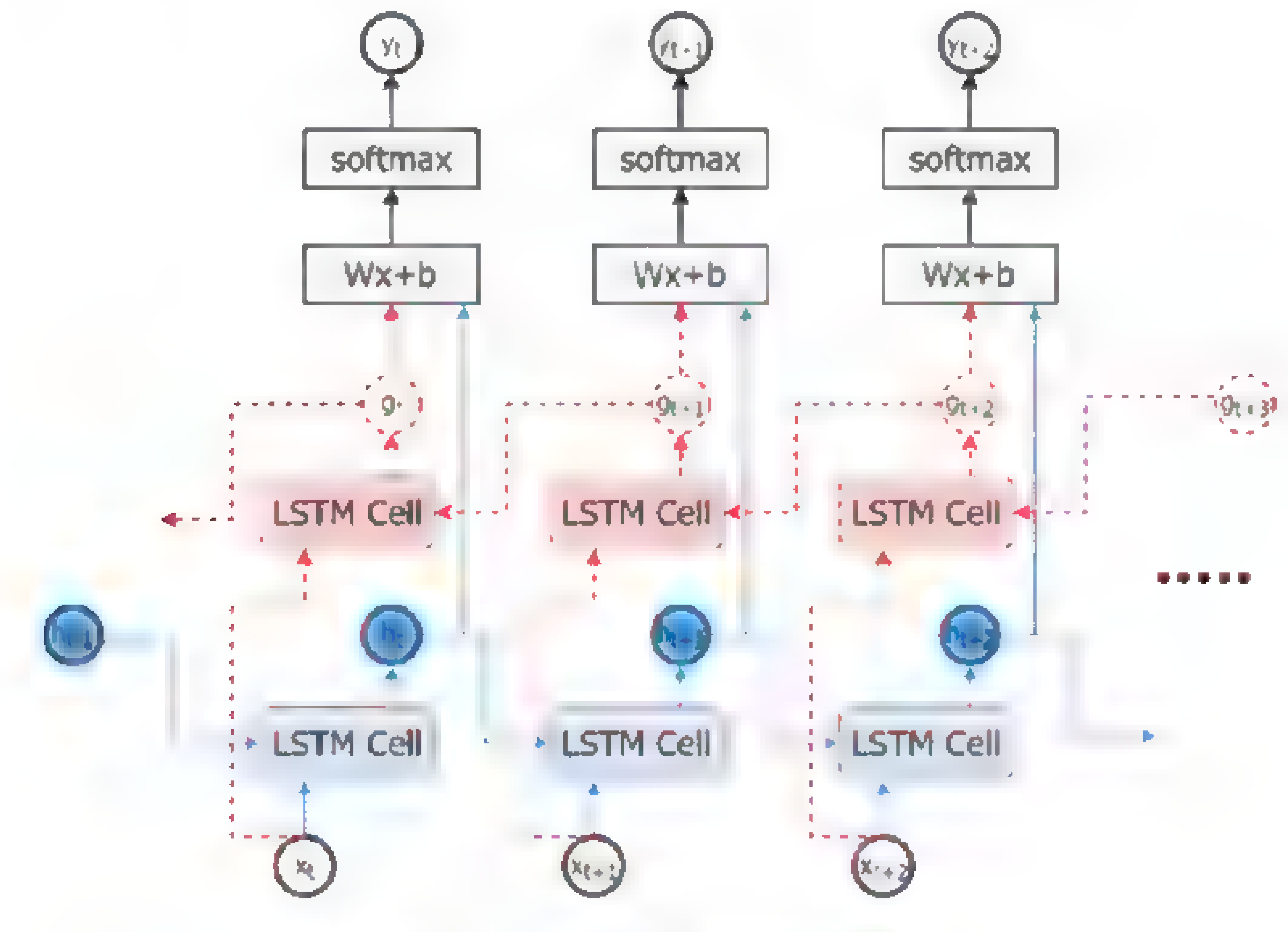


图 7-18　BiLSTM 架构示意图

训练部分有两个阶段。首先，通过从开头到结尾读取文本而创建的数据训练实线网络。该网络代表用于标准 LSTM 的正常训练流程。其次，使用通过反向读取文本生成的数据训练虚线网络。然后，在推理阶段使用实线和虚线状态的信息（通过连接两个状态并创建向量）来预测缺失单词。

7.6　LSTM 的其他变体

虽然我们主要关注标准的 LSTM 架构，但是它也有许多变体，它们既可以简化标准 LSTM 中的复杂架构，又可以产生更好的性能，或者两者兼而有之。我们将研究两种对 LSTM 细胞架构进行调整的变体：窥视孔连接（Peephole Connection）和门控循环单元（Gated Recurrent Unit，GRU）。

7.6.1　窥视孔连接

窥视孔连接不仅允许门（Gate）能够看到当前输入和先前的最终隐藏状态，还允许看到先前的细胞状态。这增加了 LSTM 细胞架构中的权重值数量。已经证明具有这种连接可以产生更好的结果，这些方程式如下：

$$i_t = \sigma(W_{(ix)}x_t + W_{(ih)}h_{(t-1)} + W_{(ic)}c_{(t-1)} + b_i)$$

$$\tilde{c}_t = \tanh(W_{(cx)}x_t + W_{(ch)}h_{(t-1)} + b_c)$$

$$f_t = \sigma(W_{(fx)}x_t + W_{(fh)}h_{(t-1)} + W_{(fc)}c_{(t-1)} + b_f)$$

$$c_t = f_t c_{(t-1)} + i_t \tilde{c}_t$$

$$o_t = \sigma(W_{(ox)}x_t + W_{(oh)}h_{(t-1)} + W_{(oc)}c_{(t)} + b_o)$$

$$h_t = o_t \tanh(c_t)$$

让我们简要介绍一下这些是如何优化 LSTM 模型的。这些门看到的是当前输入和最终隐藏状态，但不是细胞状态。不过，在此配置中，如果输出门的值接近零，即使细胞状态包含对于提升性能至关重要的信息，最终隐藏状态也将接近于零。因此，这些门在计算期间不会考虑隐藏状态。在门计算方程中含有细胞状态和被允许对细胞状态做更多的控制，并且即使在输出门接近零的情况下它也可以表现良好。

我们用图 7-19 中的窥视孔连接来说明 LSTM 的架构，标准 LSTM 中的所有现有连接进行了灰色处理，新添加的连接以黑色显示。

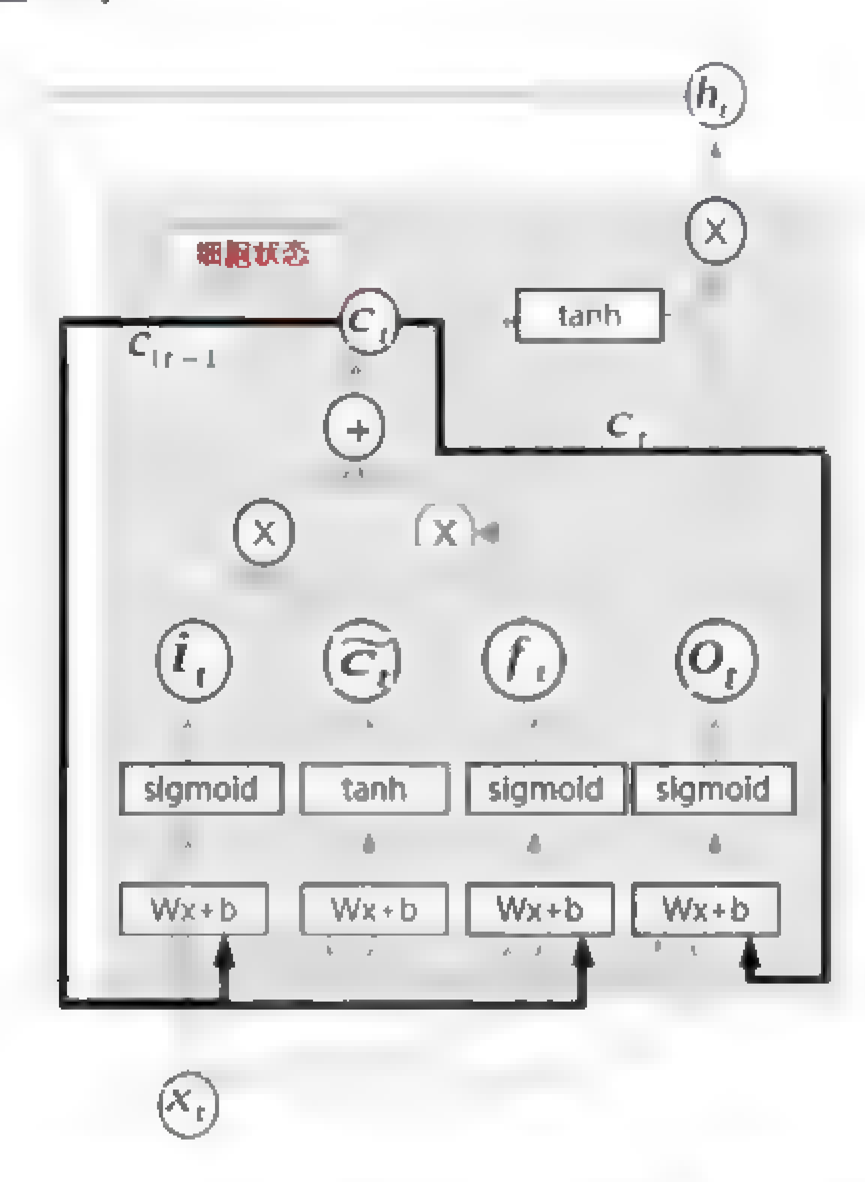


图 7-19　有窥视孔连接的 LSTM（窥视孔连接显示为黑色，其他连接显示为灰色）

7.6.2　门控循环单元

门控循环单元（Gated Recurrent Unit，GRU）可被视为标准 LSTM 架构的简化情况。正如我们看到的，LSTM 有三个不同的门和两个不同的状态。即使对于规模小的状态，仅这一点也需要大量参数。因此，科学家们研究了减少参数数量的方法，而 GRU 就是这样一种方法。标准 LSTM 和 GRU 之间有几个主要差异，具体如下：

首先，GRU 将两个状态（细胞状态和最终隐藏状态）组合成单个隐藏状态。现在，针对这两种细胞状态做简单修改后，结果就是我们失去了输出门。由前面的内容，我们知道输出门只是决定将多少细胞状态读入最终隐藏状态，因此此操作大大减少了 LSTM 细胞中的参数数量。

接下来，GRU 引入一个复位门（或称为重置门），当它接近 1 时，在计算当前状态时将获取

完整的先前状态信息；而当复位门接近 0 时，它在计算当前状态时忽略先前的状态信息。

$$r_t = \sigma(W_{(rx)}x_t + W_{(rh)}h_{(t-1)} + b_r)$$

$$\tilde{h_t} = \tanh(W_{(hx)}x_t + W_{(hh)}(r_t h_{(t-1)}) + b_h)$$

其次，GRU 将输入门和遗忘门组合成一个更新门。标准 LSTM 有两个门：输入门和遗忘门。输入门决定将多少目前输入读入细胞状态，而遗忘门决定将多少先前细胞状态读入当前细胞状态。在数学上，这可以显示如下：

$$i_t = \sigma(W_{(ix)}x_t + W_{(ih)}h_{(t-1)} + b_i)$$

$$f_t = \sigma(W_{(fx)}x_t + W_{(fh)}h_{(t-1)} + b_f)$$

如果更新门是 0，就将先前细胞状态的完整状态信息推入当前细胞状态，其中当前输入没有被读入当前状态。如果更新门是 1，那么所有当前输入被读入当前细胞状态，而之前细胞状态都不会被推送到当前细胞状态。换句话说，更新门变为遗忘门的反转，即$1-f_t$：

$$z_t = \sigma(W_{(zx)}x_t + W_{(zh)}h_{(t-1)} + b_z)$$

$$h_t = z_t\tilde{h_t} + (1 - z_t)h_{(t-1)}$$

现在我们把 GRU 设计的方程式总结一下，具体如下：

$$r_t = \sigma(W_{(rx)}x_t + W_{(rh)}h_{(t-1)} + b_r)$$

$$\tilde{h_t} = \tanh(W_{(hx)}x_t + W_{(hh)}(r_t h_{(t-1)}) + b_h)$$

$$z_t = \sigma(W_{(zx)}x_t + W_{(zh)}h_{(t-1)} + b_z)$$

$$h_t = z_t\tilde{h_t} + (1 - z_t)h_{(t-1)}$$

这比 LSTM 更紧凑。在图 7-20 中，我们给出了 GRU 细胞（左图）和 LSTM 细胞（右图）。

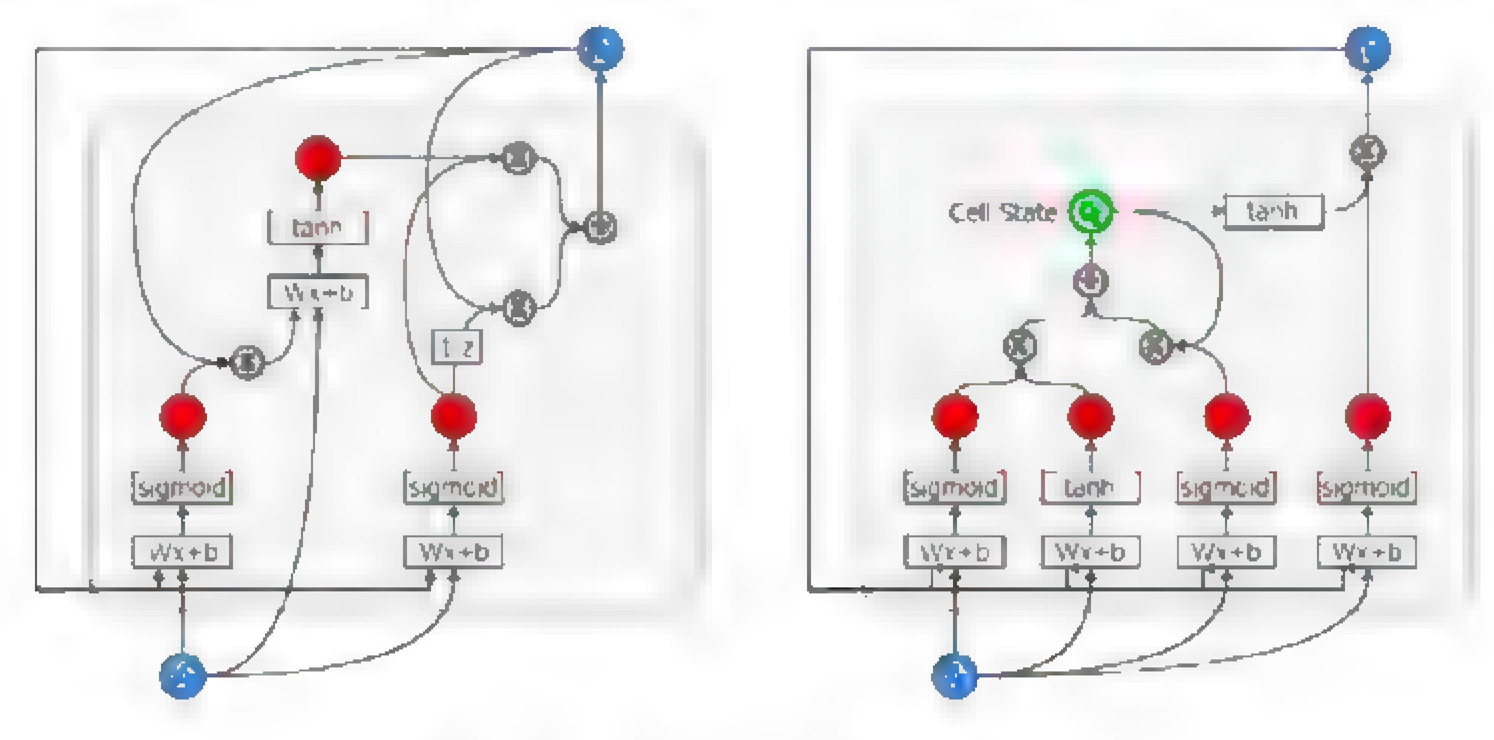


图 7-20　GRU 和 LSTM 的对比

7.7 总结

在本章中，我们学习了 LSTM 网络。首先，我们讨论了 LSTM 及其高级架构，还深入研究了 LSTM 中发生的详细计算，并通过一个例子讨论了计算过程。

我们看到 LSTM 主要由 5 个不同的部分组成。

- 细胞状态（Cell State）：LSTM 细胞的内部状态（存储器）。
- 隐藏状态（Hidden State）：外部隐藏状态用于计算预测。
- 输入门（Input Gate）：这决定了当前输入被读入细胞状态的程度。
- 遗忘门（Forget Gate）：这确定了将先前的细胞状态发送到当前细胞状态的程度。
- 输出门（Output Gate）：这决定了将多少细胞状态输出到隐藏状态。

拥有如此复杂的结构允许 LSTM 很好地捕获短期和长期依赖性。

我们发现 LSTM 实际上能够将长期依赖性作为其结构的固有部分进行学习，而 RNN 可能无法学习长期依赖性。之后，我们讨论了 LSTM 如何通过其复杂的结构解决梯度消失的问题。

然后，我们讨论了几个可以提高 LSTM 性能的扩展。首先是我们称之为贪婪采样的一种非常简单的技术，其中，我们不是总是输出最佳候选项，而是从一组最佳候选项中随机采样预测。我们看到这种技术改善了生成文本的多样性。接下来，我们研究了一种称为集束搜索的更复杂的搜索技术。通过这种技术，我们可以预测未来的几个时间步长，并选择产生最佳联合概率的候选项，而不是对未来的单个时间步长进行预测。另一项改进是词向量如何帮助我们提高 LSTM 预测的质量。使用词向量，LSTM 可以更有效地学习在预测期间替换语义相似的词（例如，代替输出 dog，LSTM 可能输出 cat），从而使生成文本更真实和正确。我们考虑的最终扩展是 BiLSTM 或双向 LSTM。BiLSTM 的一个流行应用是填充短语中的缺失单词。BiLSTM 可以从开始到末端、从末端到开始两个方向读取文本。这为我们提供了更多的文本背景资料，因为我们在正式预测之前可以观察到待生成文本的过去和未来。

最后，我们讨论了 LSTM 的两种变体：窥视孔连接和 GRU。Vanilla LSTM 在计算门时，只查看当前输入和隐藏状态。通过窥视孔连接，我们使门计算依赖于所有当前输入、隐藏和细胞状态。

GRU 是 VanillaLSTM 更优秀的变体，可简化 LSTM 而不会影响性能。GRU 只有两个门和一个状态，而 VanillaLSTM 有三个门和两个状态。

在下一章中，我们将看到这些不同的架构与每个架构的实现，并了解它们在文本生成任务中的表现。

第8章

利用 LSTM 自动生成文本

文本自动生成是自然语言处理领域的一个重要研究方向，这项技术可以实现让计算机像人类一样撰写出高质量的自然语言文本。就不同的输入而言，我们可以将文本自动生成划分为文本到文本的生成（Text-to-Text Generation）、意义到文本的生成（Meaning-to-Text Generation）、数据到文本的生成（Data-to-Text Generation）以及图像到文本的生成（Image-to-Text Generation）等，其中图像到文本生成方面的研究内容将在下一章单独介绍。这些文本生成技术每一项都有很大的挑战性，有不少研究人员都在做着前沿的研究，且相关领域应用的潜力巨大。例如，文本自动生成技术可以应用于基于海量语料库的智能问答和对话、新闻文章的撰写、突发事件的实时报道以及海量文献书籍的自动摘要等，甚至可以用来帮助学者进行学术论文的撰写，从而实现更加智能和自然的人机交互，以减少许多语言工作人员的工作量，并推动相关领域向更高的水平发展。比较典型的案例：美联社 2014 年 7 月已采用新闻写作软件自动撰写新闻稿件来报道公司业绩，美国洛杉矶时报利用应用软件来撰写突发新闻，美国“自动洞察力”公司（Automated Insights）采用“语言专家”软件撰写橄榄球、财经等方面的新闻报道。这些具体案例标志着文本自动生成技术已经对我们的工作和生活产生了积极的影响。

在第 7 章，我们已经很好地理解了 LSTM 的基本原理，例如它们如何解决梯度消失和更新规则的问题，下面可以看看如何在 NLP 任务中使用它们。 LSTM 大量用于文本生成和图像标题生成等任务。例如，语言建模对于文本摘要任务非常有用，或者为产品生成迷人的文本广告，其中图像标题生成或图像注释对于图像检索非常有用，并且用户可能需要检索表示某些概念的图像（例如一只狗）。

本章先从文本自动生成的前三个技术开始讲解，接着介绍使用 LSTM 生成新文本。为此，我们将下载格林兄弟的一些民间故事的文本集，将使用这些故事来训练 LSTM，并在最后要求输出一个全新的故事。我们将通过把文本分解成字符级二元组（n-gram，其中 n=2）来处理文本，并利用唯一的二元组来制作词汇表。我们还将探索实现之前提到的技术和方法，例如贪婪采样或集束搜索预测。之后，我们将介绍如何实现除标准 LSTM 之外的时间序列模型，例如具有窥视孔连接和 GRU

的 LSTM。

接下来，我们将学习生成具有比字符级二元组（比如单个单词）更好的输入表示的文本。注意，与字符级二元组相比，具有 One-Hot 编码的单词特征是非常低效，因为词汇量可以随着单词快速增长。因此，解决这个问题的一个好方法是学习词向量（或使用预处理的向量），并将其作为 LSTM 的输入。使用词向量能够避免维数灾难。在现实世界的这类问题中，词汇量的大小可以介于 10 000~1 000 000 之间。然而，尽管词汇量很大，不过词向量具有固定的维度。

8.1 文本到文本的生成

文本到文本的生成技术旨在对给定文本进行变换和处理，从而获得新文本，涵盖文本摘要（Document Summarization）、句子压缩（Sentence Compression）、句子融合（Sentence Fusion）、文本复述生成（Paraphrase Generation）等主要方面。对这些不同的技术，研究人员已经进行了很多年的研究，且获得的研究成果大多在自然语言处理相关学术会议与期刊上进行了公开发表，例如 ACL、EMNLP、NAACL、COLING、AAAI、IJCAI、SIGIR、INLG、ENLG 等。实际上，从某种程度上而言，我们可以将机器翻译当作一种从源语言到目标语言的文本生成技术。当然，由于机器翻译是相对独立的一个研究领域，因此我们将在第 11 章针对这方面的内容进行单独介绍。

8.1.1 文本摘要

文本摘要技术是通过自动分析给定的文档或文档集，抽取其中的要点信息，进而输出一篇短小的摘要（通常包含几句话或上百字），该摘要中的内容可直接出自原文，也可重新撰写生成。我们进行文本摘要的目的是通过对原文本进行压缩、提炼，为用户提供简明扼要的内容描述。关于文本摘要，业界有着不同的划分标准，具体如下：

根据所要处理的文档数量不同，文本摘要可以分为单文本摘要和多文本摘要。显然，单文本摘要只对单篇文本生成摘要，多文本摘要则对一个文本集生成摘要。

根据提供的上下文语境不同，文本摘要可以分为主题或查询无关的摘要和主题或查询相关的摘要。主题或查询无关的摘要是指不给定主题和查询的条件下对文本或文本集生成的摘要；主题或查询相关的摘要是指在给定的某个主题或查询的情况下能够阐明该主题或回答该查询。

根据摘要所采用的方法不同，文本摘要可以分为生成式和抽取式。生成式一般需要利用自然语言理解技术对文本进行语法和语义方面的分析，并对信息进行融合，利用自然语言生成技术生成新的摘要内容。而抽取式则相对比较简单，主要利用不同方法对文档结构单元（句子、段落等）进行评价，对每个结构单元赋予一定权重值，然后选择最重要的结构单元组成摘要。应用比较多的是抽取式，通常采用的结构单元为句子。

根据摘要的应用类型不同，摘要可以分为标题摘要、传记摘要、电影摘要等。这些摘要一般是为了满足某些特定的应用需求，比如传记摘要的目的是为某个人生成一个概括性的描述，一般涉及这个人的各种属性：姓名、性别、地址、出生、兴趣爱好等。我们通过浏览某个人的传记摘要就可

以对这个人有整体上的认知。

目前，大多数文本摘要方法主要是基于句子抽取，即以原文中的句子为单位进行评估与抽取。这类方法易于实现且能最大程度地保证摘要句子具有良好的可读性。这类方法的两个关键步骤为：一是对文档中的句子进行重要性计算或排序，二是筛选出重要的句子组合成所需的摘要。关于第一个步骤，我们可以利用基于规则的方法，通过句子的位置或所包含的线索词来判定句子的重要性，也可以采用各种机器学习方法、深度学习方法，从整体上对句子的多种特征进行重要性的分类、回归或排序。对于第二个步骤，则基于第一个步骤的结果，需要考虑句子之间的相似性，避免选择重复的句子（如 MMR 算法），并进一步对所选择的摘要句子进行连贯性排列（如自底向上法），从而获得最终的摘要。最近，学术界进一步提出了基于整数线性规划的方法以及次模函数最大化的方法，可以在句子选择的过程中同时考虑句子的冗余性。

而压缩式文本摘要方法则不同，它考虑对句子进行压缩，以便在较短长度限制下让摘要涵盖更多的内容。这里代表性的做法是同时进行句子选择与句子压缩，能够获得更好的 ROUGE 性能（注：ROUGE 是由 Lin 和 Hovy 提出的一种自动摘要评估方法，被广泛用于摘要评测任务中）。另外，部分工作还利用句子融合等技术来对已有句子进行转换，获取新的摘要内容。

关于进一步的生成式摘要研究，就是通过对原文档进行语义理解，将原文档表示为深层语义形式，然后分析获得摘要的深层语义表示，最终由摘要的深层语义表示来生成摘要文本。其中的一种尝试就是基于抽象意义表示（Abstract Meaning Representation，AMR）发展出生成式摘要。我们通过这类方法所获得的摘要句子并不是基于原文句子，而是利用自然语言生成技术从语义表达直接生成。实际上，这类方法相对比较复杂，这里还会涉及自然语言理解与自然语言生成本身的问题，具有一定的难度。

8.1.2 句子压缩与融合

句子压缩与融合技术主要用于文本摘要系统中，以便生成信息更加紧凑的摘要，使得我们获得的摘要效果更佳。具体来看，句子压缩技术主要是基于一个长句子来生成一个短句子，且保证该短句子能够保留长句子中的重要信息，即重要信息基本不丢失、短句子保持通顺。句子融合技术则将两个或多个包含重复内容的相关句子合并起来获得一个句子。这里根据目的的不同可以分为两类，一类句子融合是只保留多个句子中的共同信息，而过滤其中无关的细节信息（类似于集合运算中的取交集运算），另一类则是只过滤多个句子之间的重复内容（类似于集合运算中的取并集运算）。关于句子融合问题，Regina Barzilay 和 Kathleen McKeown 提出了一条流水线算法，主要涉及共同信息识别（Identification of Common Information）、融合网格计算（Fusion Lattice Computation）、网格线性化（Lattice Linearization）三方面。另外，相关研究人员针对句子融合问题提出了其他的方法，其中有基于结构化辨别学习的方法、基于整数线性规划的方法和基于图的最短路径的方法等。

8.1.3 文本复述生成

文本复述生成技术是指对给定文本进行改写，以生成全新的复述文本，通常输出文本与输入文

本在表达上有所不同，但它们所表达的含义基本一致。对于复杂的文本复述生成而言，研究人员给出了基于自然语言生成的方法、基于机器翻译的方法与基于支点（Pivot）的方法等。其中，基于自然语言生成的方法是模拟人类的思维模式，先对输入的句子进行语义理解，得到该句子的语义表示，再基于得到的语义表示生成新的句子。基于机器翻译的方法是将文本复述生成问题看成是单语言机器翻译问题，并通过现有机器翻译模型为给定文本生成复述文本。基于支点的方法则是将当前语言中的输入文本翻译到另一种语言（支点），再将翻译获得的文本再次翻译回当前语言。

其实，文本复述生成技术具有广泛的应用前景，例如在机器翻译时，我们可以利用文本复述技术对复杂输入文本进行简化处理，从而提升整体翻译效果。在儿童教学领域，我们可以利用文本复述技术将晦涩难懂的文本简化处理为儿童容易理解的文本。

然而，目前能够为给定文本生成具有较小差异的复述文本，但在文本生成的质量上面临着一些挑战，主要是因为对于改写甚多的复述文本而言难以做到两点：既要做到保证其与原文本语义的一致性，又要做到该文本的可读性。

8.2 意义到文本的生成

在计算语言学领域，大多数研究人员都遵循的语义研究原则是建立在“真值条件（Truth Condition）”的基础上，认为寻找到了能够使自然语言语句成真的条件，即在某种程度上刻画了自然语言的语义。

研究人员在真值假设基础上普遍采用逻辑的方法对语义进行表征，并分别从模型论（Model Theory）和证明论（Proof Theory）两个角度进行研究，所以大多数情况下这种类型的语义也被称为逻辑语义。

其实，意义到文本的生成和组合语义分析（Compositional Semantic Parsing）密切相关，语义分析旨在对线性的词序列进行自动句法语义解析并得到其真值条件。由于我们在分析过程中遵循了弗雷格的组合原则（Principle of Compositionality），因此也称为组合语义分析，以便同分布式语义（Distributional Semantics）相区别。作为自然语言处理中的一个核心技术，组合语义在深度语义理解方面起着关键性的作用，在智能问答、机器翻译等多个自然语言处理核心任务中有着良好的应用。从问题自身的定义来看，其实意义到文本的生成与组合语义分析是一对互逆的自然语言处理任务。接下来，我们从基于深层语法的文本生成、基于同步文法的文本生成两方面做相关介绍。

8.2.1 基于深层语法的文本生成

在自然语言处理研究的早期阶段，计算机语言学起到了很大的作用，因为计算语言学家从形式化、可计算两方面对自然语言进行了建模，并提出了旨在解释语言运作机理的一系列句法语义模型，进而依据这些模型构建了自然语言处理系统。

而深层语法复杂度较高，如何构造对错综复杂的语言现象具有高覆盖度（Broad Coverage）的语法规则本身是一个极大的难题。以上研究主要是对原型算法进行讨论，而因为真实可用的大型深

层语法当时没有得到很好的开发，以上研究并没有呈现极具代表性意义的经验结果。经过十余年的开发，研究人员在 HPSG 理论的基础上开发出了英语资源语法（English Resource Grammar，ERG），这是一个比较成功的具有较高覆盖率的深层语法规则系统，而围绕 ERG 所展开的文本生成研究也取得了有益的进展。Carroll 和 Oepen 基于 ERG 和真实测试数据重新讨论了基于线图的生成技术，给出了极具参考意义的经验评估。另外，他们提出了两项新的技术来改进基于合一语法的可行解紧致表示（Compact Representation）及其相关解码算法——选择性开箱（Selective Unpacking），尤其后者有效地利用了判别式学习模型来改进文本生成过程中所遇到的歧义消解。

相关研究在二十世纪八九十年代取得了丰硕的研究成果，一系列兼具语言本体解释力和可计算性的语法范式（Grammar Formalism）被提出，如组合范畴语法（Combinatory Categorial Grammar，CCG）、中心语驱动的短语结构语法（Head-Driven Phrase-Structure Grammar，HPSG）等。不同于目前句法分析所主要使用的上下文无关文法（Context-Free Grammar，CFG），上述语法范式具有超越上下文无关的表达能力，其语法推导过程往往更复杂，蕴含更多的信息，而这些信息可以用来做更透明的语义分析，简而言之，这些深层语法范式能够更好地支持句法语义同步的语言分析。在深层语法的支撑下，通过句法语义的协同推导可以获取自然语言的组合语义，而当以语义表征作为输入，通过其逆过程可以完成意义到文本的生成。

组合范畴语法是一个广受自然语言处理领域学者关注的语法范式，其设计遵循了类型透明（Type Transparency）的原则，具有精简的语法语义接口，常常被语义分析和文本生成模型所采用。White 和 Baldridge 讨论了如何将线图生成法与组合范畴语法结合，并开发了开源的基于组合范畴语法的句子实现（Realization）工具——OpenCCG。White 又与其他学者联合提出了一些进一步改进文本生成的算法。

8.2.2 基于同步文法的文本生成

在过去的 20 年间，统计句法分析与统计机器翻译是公认的两个取得长足进步的自然语言处理技术。除了从成熟的统计句法分析中借鉴成功经验（如判别式消歧）之外，不少学者尝试复用成功的机器翻译模型来完成文本生成。机器翻译的目标是将某种自然语言语句翻译成另一种自然语言语句，并尽量保持意义不变；而文本生成则可以视为将某种形式的语言语句翻译成一种自然语言语句，二者具有极强的可比性。

Chiang 提出了基于层次短语的翻译模型（Hierarchical Phrase-Based Model），其核心是利用同步上下文无关文法（Synchronous Context-Free Grammar）来协同源语言语句的解析和目标语言语句的生成。目前同步文法已经被借鉴到文本生成的研究中。Wong 与 Mooney 两位作者讨论了两种形式语言用于表征意义：第一种是用于指挥机器人动作的形式语言，第二种是无变量的数据库检索语言；而 Lu 与 Ng 则针对表达能力极强的类型 λ 表达式（Typed λ-expression）展开研究。两项研究的一个共同点是构建形式语言的基于树的结构，再将相关结构与待生成的自然语言的树结构建立一致性对应，从而完成文本生成任务；另一个共同点则是广泛地使用现有的机器翻译技术（包括开源软件等）来进行文法抽取、解码等。

国内语言学界与计算语言学界针对自然语言语义的形式化研究较少，针对汉语进行全方面组合

语义刻画的研究目前尚属空白。从事自然语言处理的研究人员也较少涉猎深层语言结构处理问题，而对意义到文本的生成研究则更是鲜有，很少能见到相关学术成果发表在重要学术会议和期刊上。

8.3 数据到文本的生成

数据到文本的生成技术指根据给定的数值数据生成相关文本，例如基于数值数据生成天气预报文本、体育新闻、财经报道、医疗报告等。数据到文本的生成技术具有极强的应用前景，目前该领域已经取得了很大的研究进展，业界已经研制出面向不同领域和应用的多个生成系统。针对数据到文本的生成技术研究主要集中于少数几个单位，例如英国阿伯丁大学、英国布莱顿大学、爱丁堡大学等，相关研究成果主要发表在 INLG、ENLG 等专业学术会议上。

英国阿伯丁大学的 Ehud Reiter 在三阶段流水线模型的基础上提出了数据到文本的生成系统的一般框架，如图 8-1 所示。

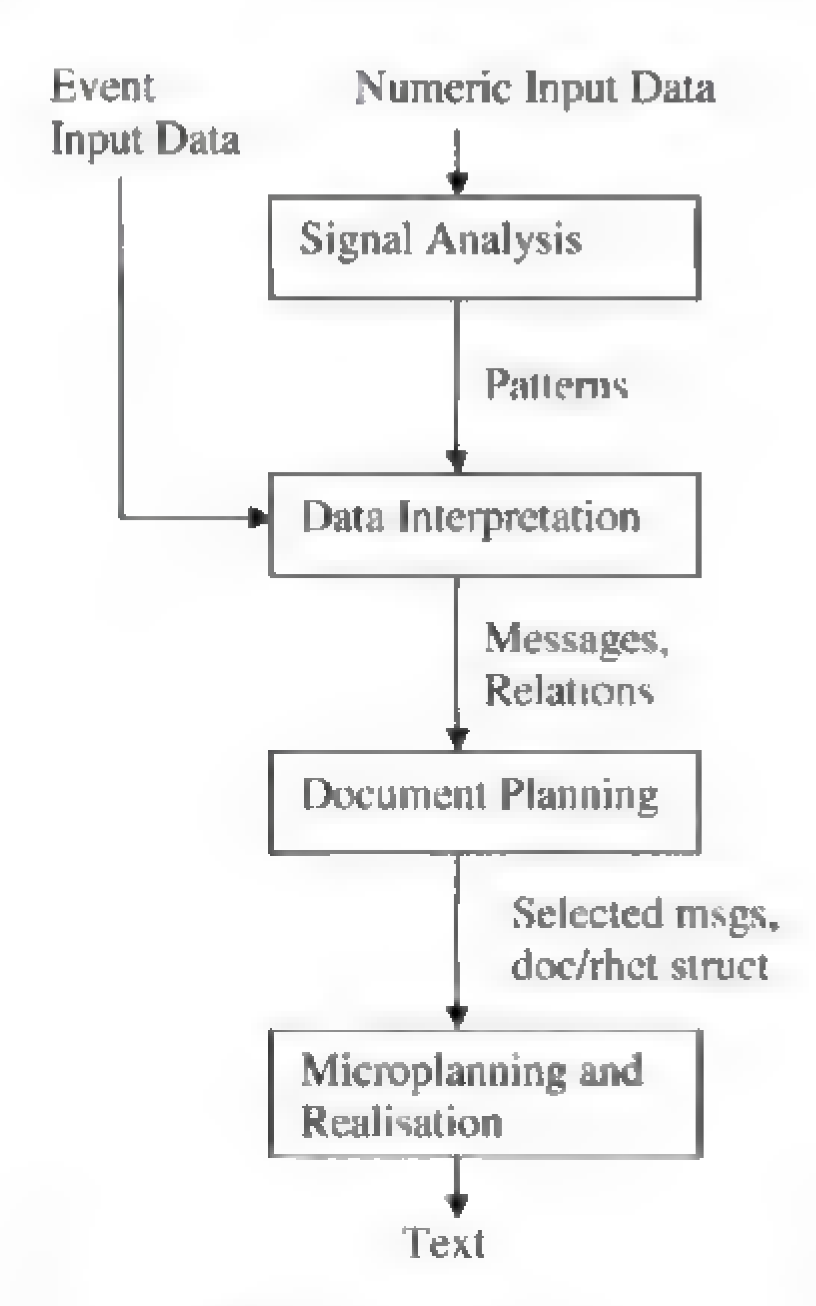


图 8-1 数据到文本的生成系统的一般框架

其中：

信号分析（Signal Analysis）模块的输入为数值数据，通过利用各种数据分析方法检测数据中的基本模式，输出离散数据模式。例如股票数据中的峰值、较长期的增长趋势等。该模块与具体应用领域和数据类型相关，针对不同的应用领域与数据类型输出的数据模式是不同的。

数据阐释（Data Interpretation）模块的输入为基本模式与事件，通过对基本模式和输入事件进行分析而推断出更加复杂和抽象的消息，同时推断出它们之间的关系，最后输出高层消息以及消息之间的关系。例如针对股票数据，如果跌幅超过某个值就可以创建一条消息。还需要检测消息之间

的关系，例如因果关系、时序关系等。值得说明的是，数据阐释模块并不是在所有文本生成系统中都需要，例如在天气预报文本生成系统中，基本的模式足以满足要求，因此并不需要采用数据阐释模块。

文档规划（Document Planning）模块的输入为消息及关系，分析决定哪些消息和关系需要在文本中提及，同时要确定文本的结构，最后输出需要提及的消息以及文档结构。从更高的层次来说，信号分析与数据阐释模块会产生大量的消息、模式和事件，但文本通常长度受限，只能描述其中的一部分，因此文档规划模块必须确定文本中需要说明的消息。一般可根据专家知识、消息的重要性及新颖性等来进行选择和确定。当然，该模块与领域也相关，不同领域中对消息的选择所考虑的因素不一样，文档的结构也会不一样。

微规划与实现（Microplanning and Realization）模块的输入为选中的消息及结构，通过自然语言生成技术输出最终的文本。该模块主要涉及对句子进行规划以及句子的实现，要求最终实现的句子具有正确的语法、时态和拼写，同时采用准确的指代表达。

目前，业界已经研制了面向多个领域的数据到文本的生成系统，这些系统的框架与上述一般框架并无大的差别，部分系统将上述框架中的两个模块合并为一个模块，或者省去其中一个模块。数据到文本的生成技术在天气预报领域应用得最为成功，业界研制了多个系统对天气预报数据进行总结，生成天气预报文本。例如，FoG 系统能够从用户操作过的数据中生成双语天气预报文本；SumTime 系统能够生成海洋天气预报文本，实验评测表明用户有时更倾向于阅读 SumTime 所生成的天气预报，而非专家撰写的天气预报。此外，英国阿伯丁大学的 Anja Belz 提出了概率生成模型进行天气语言文本的生成。Anja Belz 和 Eric Kow 进一步基于天气预报数据分析对比了多种数据到文本的生成系统，结果表明采用自动化程度较高的方法并不会降低文本生成质量，同时文本质量的自动评价方法会低估基于手工规则构建的系统，而高估自动化系统。

由于数据到文本的生成技术的巨大应用价值，业界成立了多家从事文本生成的公司，能够为多个行业基于行业数据生成行业报告或新闻报道，从而节省大量的人力。比较知名的公司有 ARRIA、AI（Automated Insights）、Narrative Science 等。其中 ARRIA 是一家总部设在欧洲的公司，其前称为 Data2Text，由来自阿伯丁大学的两名教授 Ehud Reiter 与 Yaji Sripada 创办，后来自然语言生成领域的另一位科学家 Robert Dale 也加入了该公司，该公司的核心技术为 ARRIA NLG 引擎。AI 则是一家美国人工智能公司，由一名思科的前工程师 Robbie Allen 创办，最早基于体育数据生成文本摘要，目前能为包括金融、个人健身、商业智能、网站分析等在内的多个领域的数据生成文本报告，其核心技术为 WordSmith NLG 引擎。目前，AI 公司已经为美联社等多家单位生成数亿篇新闻报道，造成了巨大的影响力。Narrative Science 则是根据美国西北大学的一个研究项目 StatsMonkey 发展而来的，其核心技术为 Quill NLG 引擎。Forbes 是 Narrative Science 的一个典型客户，在网站上有一个 Narrative Science 专页，全部文章都是由 Narrative Science 自动生成的。下面给出一篇自动生成的样例新闻，如图 8-2 所示。

Earnings for Alcoa Projected to Rise

By Narrative Science

• Comment Now • Follow Comments

Wall Street is high on **Alcoa**, expecting it to report earnings that are up 28% from a year ago when it reports its second-quarter earnings on Wednesday, July 8, 2015. The consensus estimate is 23 cents per share, up from earnings of 18 cents per share a year ago.

The consensus estimate has fallen over the past three months, from 27 cents. Analysts are expecting earnings of 95 cents per share for the fiscal year. Analysts look for revenue to decrease 1% year-over-year to $5.79 billion for the quarter, after being $5.84 billion a year ago. For the year, revenue is projected to roll in at $23.63 billion.

Revenue dropped year-over-year in the first quarter, ending a two-quarter streak of growing revenue.

Alcoa is a global producer of aluminum. It is mainly engaged in the production and management of primary aluminum, fabricated aluminum, and alumina combined. It is actively involved in a range of industries, including technology, mining, smelting, and recycling. Kaiser Aluminum Corp., also in the metal mining industry, will report earnings on Wednesday, July 22, 2015. Analysts are expecting earnings of $1.19 per share for Kaiser Aluminum, up 13% from last year's earnings of $1.05 per share. Other companies in the metal mining industry with upcoming earnings release dates include: Noranda Aluminum Holding and Aluminum Corp. of China Limited.

Earnings estimates provided by Zacks.

图 8-2 Narrative Science 自动生成的样例新闻

接下来，我们将着手利用 LSTM 模型逐步实现文本自动生成项目。

8.4 文本自动生成前的数据准备

首先，我们将讨论用于文本生成的数据以及用于清理数据的各种预处理步骤。

8.4.1 数据集

首先，我们看一下数据集的样子，以便当看到生成的文本时，根据训练数据评估它是否有意义。我们将从网站 https://www.cs.cmu.edu/~spok/grimmtmp/ 上下载前 100 本书。这是格林兄弟的一套书（从德语到英语）的译文，与第 6 章循环神经网络中用于证明 RNN 性能的文本相同。

首先，使用脚本自动从网站下载前 100 本书，代码如下：

```
url = 'https://www.cs.cmu.edu/~spok/grimmtmp/'
# 如果有必要的话，创建一个目录
dir_name = 'stories'
```

```
    if not os.path.exists(dir_name):
        os.mkdir(dir_name)
    def maybe_download(filename):
"""如果文件不存在就下载"""
        print('Downloading file: ', dir_name+ os.sep+filename)
        if not os.path.exists(dir_name+os.sep+filename):
            filename, _ = urlretrieve(url + filename, dir_name+os.sep+filename)
        else:
          print('File ',filename, ' already exists.')
        return filename
    num_files = 100
    filenames = [format(i, '03d')+'.txt' for i in range(1,101)]
    for fn in filenames:
        maybe_download(fn)
```

我们将展示从两个随机挑选的故事中提取的示例文本片段。

以下是第一个文本片段（0.01.txt）：

```
The king's daughter began to cry, for she was afraid of the cold frog
which she did not like to touch, and which was now to sleep in her
pretty, clean little bed. But the king grew angry and said, "He who
helped you when you were in trouble ought not afterwards to be
despised by you." So she took hold of the frog with two fingers,
carried him upstairs, and put him in a corner, but when she was in
bed he crept to her and said, "I am tired, I want to sleep as well as
you, lift me up or I will tell your father." At this she was terribly
angry, and took him up and threw him with all her might against the
wall. "Now, will you be quiet, odious frog," said she. But when he
fell down he was no frog but a king's son with kind and beautiful
eyes. He by her father's will was now her dear companion and
husband. Then he told her how he had been bewitched by a wicked
witch, and how no one could have delivered him from the well but
herself, and that to-morrow they would go together into his kingdom.
```

第二个文本片段（0.02.txt）如下：

```
Next morning, when the loss was reported abroad, all the people cried
loudly 'the queen is a man-eater. She must be judged, and the king
was no longer able to restrain his councillors. Thereupon a trial was
held, and as she could not answer, and defend herself, she was
condemned to be burnt at the stake. The wood was got together, and
when she was fast bound to the stake, and the fire began to burn
round about her, the hard ice of pride melted, her heart was moved by
repentance, and she thought 'if I could but confess before my death
```

```
that I opened the door.' Then her voice came back to her, and she
cried out loudly 'yes, mary, I did it, and straight way rain fell
from the sky and extinguished the flames of fire, and a light broke
forth above her, and the virgin mary descended with the two little
sons by her side, and the new-born daughter in her arms.  She spoke
kindly to her, and said 'he who repents his sin and acknowledges it,
is forgiven.' Then she gave her the three children, untied her
tongue, and granted her happiness for her whole life.
```

8.4.2 预处理数据

在预处理方面，我们首先使所有文本字母变为小写，并将文本分解成字符 n-gram，其中 n=2。请考虑以下句子：

```
The king was hunting in the forest.
```

这将分解为一系列 n-gram，如下所示：

```
['th,' 'e ,' 'ki,' 'ng,' ' w,' 'as,' …]
```

我们将使用字符级别的双字母，因为与使用单个单词相比，它大大减少了词汇量。此外，我们将使用特殊标记（UNK）替换在语料库中出现少于 10 次的所有双字母组，表示该二元组未知。这有助于我们进一步减少词汇量。

8.5 实现 LSTM

本节将讨论 LSTM 实现的细节。虽然 TensorFlow 中有子库已经实现了现成的 LSTM，但我们将从头开始实现。这将非常有价值，因为在现实世界中可能存在无法直接使用这些现成组件的情况。此代码位于 ch8 文件夹中的 8_lstm_for_text_generation.ipynb 代码文件中。但是，我们还将包含一个范例，其中将展示如何使用现有的 TensorFlow RNN API，该 API 在位于同一文件夹中的 8_lstm_word2vec_rnn_api.ipynb 代码文件中。在这里，我们将讨论 8_lstm_for_text_generation.ipynb 文件中可用的代码。

首先，我们将讨论用于 LSTM 的超参数及其效果。其次，将讨论实现 LSTM 所需的参数（权重值和偏差）。接着，将讨论如何使用这些参数来编写 LSTM 中发生的操作。之后，我们将了解如何逐步地将数据提供给 LSTM。接下来，将讨论如何使用梯度裁剪实现参数以进行优化。最后，我们将研究如何使用学习到的模型输出预测，这些预测基本上是二元组，它们最终将组成一个有意义的故事。

8.5.1　定义超参数

首先，我们将定义 LSTM 所需要的一些超参数：

```
# 隐藏状态变量中的神经元数量
num nodes = 128
#  一个 batch 中要处理的数据点数
batch size = 64

# 在优化期间展开的时间步数
num_unrollings = 50
#使用 dropout
dropout = 0.2
```

以下给出了每个超参数的说明。

- num_nodes：表示细胞（Cell）记忆状态中的神经元数量。当数据充足时，增加细胞记忆的复杂性将给出更好的性能，与此同时，它减慢了计算速度。
- batch_size：这是一步处理的数据量。增加批量的大小有助于提高模型的性能，但会带来更高的内存要求。
- num_unrollings：在截断 BPTT 中使用的时间步数。num_unrollings 步数越高，性能越好，但是它将增加内存需求和计算时间。
- dropout：使用 dropout（正则化技术）来减少模型的过拟合，并产生更好的结果。dropout 在将输入、输出、状态变量传递到它们的连续操作之前，随机地从输入、输出、状态变量中删除信息。这在学习过程中产生冗余特征，从而导致更好的性能。

8.5.2　定义参数

现在我们将为 LSTM 的实际参数定义 TensorFlow 变量。

首先，我们将定义输入门参数。

- ix：这些是将输入连接到输入门的权重值。
- im：这些是将隐藏状态连接到输入门的权重值。
- ib：这是偏差项。

这里我们将定义参数：

```
#Input gate (it)  - 写入细胞状态的记忆量
#将当前输入连接到输入门
ix = tf.Variable(tf.truncated_normal([vocabulary_size, m_nodes],stddev=0.02))
# 将先前的隐藏状态连接到输入门
im = tf.Variable(tf.truncated normal([num nodes, num nodes],stddev=0.02))
# 输入门的偏差
```

```
ib - tf.Variable(tf.random uniform([1, num nodes], 0.02, 0.02))
```

类似的，我们将为遗忘门、候选项（用于存储器单元数的计算）和输出门定义权重值。

遗忘门定义如下：

```
#遗忘门 (ft) - 从细胞状态丢弃多少记忆
#将当前输入连接到遗忘门
fx = tf.Variable(tf.truncated normal([vocabulary size, num nodes],stddev=0.02))
#将之前的隐藏状态连接到遗忘门
fm = tf.Variable(tf.truncated normal([num nodes, num nodes],stddev=0.02))
# 遗忘门的偏差
fb = tf.Variable(tf.random_uniform([1, num_nodes],-0.02, 0.02))
```

候选项（用于计算细胞状态）定义如下：

```
#Candidate value (c~t) - 用于计算当前的细胞状态
#将当前输入连接到候选项
cx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],stddev=0.02))
#将之前的隐藏状态连接到候选项
cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],stddev=0.02))
# 候选项的偏差
cb = tf.Variable(tf.random_uniform([1, num_nodes],-0.02,0.02))
```

输出门定义如下：

```
#输出门 - 从细胞状态输出多少记忆
#将当前输入连接到输出门
ox = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes],stddev=0.02))
#将先前的隐藏状态连接到输出门
om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes],stddev=0.02))
#输出门的偏差
ob = tf.Variable(tf.random_uniform([1, num_nodes],-0.02,0.02))
```

接下来，我们将为状态和输出定义变量。它们是 TensorFlow 变量，表示 LSTM 细胞的内部状态和外部隐藏状态。在定义 LSTM 计算的操作时，我们使用 tf.control_ies(...)函数定义这些操作，以便用我们计算的最新细胞状态和隐藏状态值更新。

```
    #变量在展开时保存状态
    #隐藏状态
    saved_output = tf.Variable(tf.zeros([batch_size, num_nodes]),trainable=False,
name='train_hidden')
    # 细胞状态
    saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]),trainable=False,
name='train_cell')
    #验证阶段的相同变量
    saved valid output = tf.Variable(tf.zeros([1, num nodes]),trainable False,
```

```
name 'valid hidden')
    saved valid state = tf.Variable(tf.zeros([1, num nodes]),trainable-False,
name-'valid cell')
```

最后，我们将定义 softmax 层以获得实际预测：

```
    #Softmax 分类器的权重值和偏差
    w = tf.Variable(tf.truncated normal([num nodes,
vocabulary size],stddev=0.02))
    b = tf.Variable(tf.random uniform([vocabulary size],-0.02,0.02))
```

注意，我们使用的是正态分布，具有零均值和小的标准偏差。这是很好的，因为我们的模型是一个简单的单个 LSTM 细胞。然而，当网络变得更深时（多个 LSTM 细胞堆叠在一起），需要更精细化的初始化技术。一种这样的初始化技术被称为 Xavier 初始化，由 Glorot 和 Bengio 在他们的论文《*Understanding the difficulty of training deep feedforward neural networks*》中提出（第 13 届国际人工智能与统计计会议论文集，2010 年）。这可以作为 TensorFlow 中的变量初始化器使用（https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer）。

8.5.3 定义 LSTM 细胞及其操作

通过定义权重值和偏差可以在 LSTM 细胞中定义操作，这些操作包括以下内容：

- 计算输入门和遗忘门产生的输出。
- 计算内部细胞状态。
- 计算输出门产生的输出。
- 计算外部隐藏状态。

以下是 LSTM 细胞的实现情况：

```
    def lstm_cell(i, o, state):
         input_gate = tf.sigmoid(tf.matmul(i, ix) +tf.matmul(o, im) + ib)
         forget_gate = tf.sigmoid(tf.matmul(i, fx) +tf.matmul(o, fm) + fb)
         update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
         state = forget_gate * state + input_gate * tf.tanh(update)
         output_gate = tf.sigmoid(tf.matmul(i, ox) +tf.matmul(o, om) + ob)
         return output_gate * tf.tanh(state), state
```

8.5.4 定义输入和标签

现在我们将定义训练输入（展开训练）和标签。训练输入是具有 num unrolling 批量数据（序列）的列表，其中每批数据具有[batch size，vocabulary size]大小：

```
    train inputs, train labels = [],[]
```

```
for ui in range(num unrollings):

train inputs.append(tf.placeholder(tf.float32,shape=[batch size,vocabulary
size],name='train inputs %d'%ui))

train labels.append(tf.placeholder(tf.float32,shape=[batch size,vocabulary
size], name = 'train labels %d'%ui))
```

我们为验证输入和输出定义占位符，这些占位符将用于计算验证困惑度（Perplexity）。不过，在这里我们不使用输入进行与验证相关的计算。

```
# 验证数据占位符

valid_inputs = tf.placeholder(tf.float32, shape=[1,vocabulary_size],
name='valid_inputs')

valid_labels = tf.placeholder(tf.float32, shape=[1,vocabulary_size],name =
'valid_labels')
```

8.5.5 定义处理序列数据所需的序列计算

在这里，我们将以循环方式计算单次展开训练输入所产生的输出。我们还将使用 Dropout（参考 Srivastava 和 Nitish 等人的《*Dropout: A Simple Way to Prevent Neural Networks from Overfitting*》，Journal of Machine Learning Research 15（2014）：1929-1958），因为这样可以稍微提高性能。最后，通过计算的所有隐藏输出值来为训练数据计算 logit 值：

```
#训练相关的推理逻辑，在所有展开训练输入中保存计算的状态输出用于计算损失
outputs = list()

#这两个 Python 变量在展开的每个步长中迭代更新
output = saved_output
state = saved_state
#对于展开中的所有步数，循环计算隐藏状态（输出）和细胞状态（状态）
for i in train_inputs:
output, state = lstm_cell(i, output, state)
output = tf.nn.dropout(output,keep_prob=1.0-dropout)
#增加每个计算的输出值
outputs.append(output)

#计算得分值
logits = tf.matmul(tf.concat(axis 0, values outputs), w) + b
```

接下来，在计算损失之前，我们必须确保输出和外部隐藏状态被更新为之前计算的最新值。这可以通过添加 tf.control_dependencies 条件并将 logit 和 loss 计算保持在以下条件来实现：

```
    with tf.control_dependencies([saved_output.assign(output),
saved_state.assign(state)]):
    # 分类器
    loss = tf.reduce_mean( tf.nn.softmax_cross_
    entropy_with_logits_v2(logits=logits, labels=tf.concat(axis=0,
    values=train_labels)))
```

我们还定义了验证数据的前向传播逻辑。注意，我们不会在验证期间使用 dropout，仅限于训练期间：

```
    #验证阶段相关的推理逻辑
    #计算验证数据的 LSTM 细胞输出
    valid_output, valid_state = lstm_cell( valid_inputs, saved_valid_output,
saved_valid_state)
    #计算 logits
    valid_logits = tf.nn.xw_plus_b(valid_output, w, b)
```

8.5.6　定义优化器

本节将定义优化过程。我们将使用一种称为 Adam 的先进优化器，它是迄今为止最好的基于随机梯度的优化器之一。在代码中，gstep 是一个变量，用于随时间推移衰减学习速率。我们将在下一节讨论细节。此外，我们将使用梯度裁剪来避免梯度爆炸。

```
    #每次 gstep 增加时衰减学习率
    tf_learning_rate = tf.train.exponential_decay(0.001,gstep,decay_steps=1,
decay_rate=0.5)
    # Adam 优化器和梯度裁剪
    optimizer = tf.train.AdamOptimizer(tf_learning_rate)
    gradients, v = zip(*optimizer.compute_gradients(loss))
    gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
    optimizer = optimizer.apply_gradients(   zip(gradients, v))
```

8.5.7　随着时间的推移衰减学习率

如前所述，我们使用衰减学习率而不是恒定的学习率。随着时间的推移，衰减学习率是深度学习中用于实现更好性能和减少过度拟合的常用技术。这里的关键思想是，如果验证困惑度在预定数量的时期内没有减少，就降低学习速率（例如降低 0.5 倍）。让我们来看一下这是如何实现的。

首先定义 gstep 和一个增加 gstep 的操作，名为 inc_gstep，代码如下：

```
    #学习率衰减
```

```
gstep = tf.Variable(0,trainable False,name 'global step')
#运行此操作将导致 gstep 的值增加，从而降低学习速率
inc gstep = tf.assign(gstep, gstep+1)
```

通过这个定义可以编写一些简单的逻辑在验证损失没有减少的情况下调用 inc gstep 操作，代码如下：

```
#学习率衰减相关
# 如果验证数据的困惑度没有持续减少，那么这个时期会降低学习率
   decay threshold = 5
   # 保持计数困惑度增加
   decay_count = 0
   min_perplexity = 1e10
   # 学习率衰减逻辑
   def decay_learning_rate(session, v_perplexity):
        global decay_threshold, decay_count, min_perplexity
        #衰减学习率
       if v_perplexity < min_perplexity:
           decay_count = 0
           min_perplexity= v_perplexity
      else:
          decay_count += 1

      if decay_count >= decay_threshold:
         print('\t Reducing learning rate')
        decay_count = 0
        session.run(inc_gstep)
```

每当我们遇到新的最小验证困惑度时，我们都会更新 min_perplexity。此外，v_perplexity 是当前的验证困惑度。

8.5.8 进行预测

现在我们可以简单地通过对之前计算的 logits 应用 softmax 激活来进行预测。我们还定义了验证 logits 的预测操作：

```
train_prediction = tf.nn.softmax(logits)
#确保在继续下一迭代之前更新状态变量
with tf.control_dependencies([saved_valid_output.assign(valid_output),
          saved_valid_state.assign(valid_state)]):
           valid prediction = tf.nn.softmax(valid_logits)
```

8.5.9　计算困惑度（损失）

我们在第 7 章“长短期记忆网络”中定义了困惑度。回顾一下，考虑到当前的 n-gram，困惑度是 LSTM 看到下一个 n-gram 的意外（Surprised）程度。因此，较高的困惑度意味着较差的性能，而较低的困惑度意味着较好的性能：

```
train_perplexity_without_exp = tf.reduce_sum(tf.concat(train_labels,0)*
-tf.log(tf.concat(train_prediction,0)+1e-10))/(num_unrollings*batch_size)
# 计算验证数据困惑度
valid_perplexity_without_exp = tf.reduce_sum(valid_labels*-tf.
log(valid_prediction+1e-10))
```

8.5.10　重置状态

我们使用状态重置，因为正在处理多个文档。因此，在开始处理新文档时，我们将隐藏状态重置为零。但是，在实践中，重置状态是否有帮助还不是很清楚。一方面，当开始阅读新故事时，将每个文档开头的 LSTM 细胞的记忆（或者说内存）重置为零听起来很直观；另一方面，这会使状态变量偏向零。我们鼓励在状态重置和状态不重置的情况下运行算法，并查看哪种方法表现良好。

```
# 重置训练状态
reset_train_state = tf.group(tf.assign(saved_state, tf.zeros([batch_size,
num_nodes])),tf.assign(saved_output, tf.zeros([batch_size, num_nodes])))

# 重置验证状态
reset_valid_state = tf.group(tf.assign(saved_valid_state, tf.zeros([1,
num_nodes])), tf.assign(saved_valid_output, tf.zeros([1, num_nodes])))
```

8.5.11　贪婪采样打破重复性

这是一种非常简单的技术，我们可以随机采样 LSTM 从找到的 n 个最佳候选项中抽取下一个预测。

```
def sample(distribution):
        best_inds = np.argsort(distribution)[-3:]
        best_probs = distribution[best_inds]/np.sum(distribution[best_inds])
        best_idx = np.random.choice(best_inds,p=best_probs)
        return best_idx
```

8.5.12　生成新文本

我们将定义生成新文本所需的占位符、变量和操作。这些定义与我们对训练数据所做的类似。

首先，将为状态和输出定义输入占位符和变量。接下来，将定义状态的重置操作。最后，将为要生成的新文本定义 LSTM 细胞计算和预测：

```
    # 文本生成：batch 1，不展开
        test_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size], name
= 'test_input')

        # 测试阶段的相同变量
        saved_test_output = tf.Variable(tf.zeros([1, num_nodes]), trainable=False,
name='test_hidden')
        saved_test_state = tf.Variable(tf.zeros([1, num_nodes]),
trainable=False, name='test_cell')

        # 计算用于测试数据的 LSTM 细胞输出
        test_output, test_state = lstm_cell( test_input, saved_test_output,
saved_test_state)

        # 确保在继续下一次迭代之前更新状态变量
        with tf.control_dependencies([saved_test_output.assign(test_output),
            saved_test_state.assign(test_state)]):
            test_prediction = tf.nn.softmax(tf.nn.xw_plus_b(test_output, w, b))
        #重置测试状态
        reset_test_state = tf.group(
         saved_test_output.assign(tf.random_normal([1,num_nodes],stddev=0.05)),
         saved_test_state.assign(tf.random_normal([1,num_nodes],stddev=0.05)))
```

8.5.13 示例生成的文本

让我们看一下第 26 步学习后 LSTM 生成的一些数据：

```
    the little said, am one of the glass, and the father had to this, there is at
the dog
    would have should be stone with the money, hans that in his father the king was
mine, then he did not know what a beautiful bird am UNK

    thereupon they said, klippines,
        my father he did not know when you have sold his wife, and
    said, the wedding was alone.  then the money him a sunder them.
    so the shepherd
    was dead to his father, and that he had
    been he said,
    what, can you speak and said, he saw the window that he had to see him and comes,
he fell into a son, and
```

```
    went out and where that
    this all heads which he did better, he were was to standing by the handsome of
the brother, said the wine of his strang the way to the faith, and said, you have
promised her this, said, the little marlinchen by his father, i will not know how
i will have you are, and they were to her father, i will now change man, and
    wept again, and the father, and the master loses.
    so the master the good brother was a couple in
```

正如你所看到的，文本看起来比我们从 RNN 生成的要好得多。事实上，我们的训练语料库里有一个关于农妇和奶牛贩子的故事。但是，我们的 LSTM 不只是输出文本，还可以引入新的东西比如牧羊人（shepherd）、婚礼（wedding）和葡萄酒（wine）为这个故事情节增添更多的色彩。接下来，我们将研究从标准 LSTM 生成的文本与其他模型（如带窥视孔连接和 GRU 的 LSTM）生成的文本的比较。

8.6 标准 LSTM 与带有窥视孔连接和 GRU 的 LSTM 的比较

本节将在文本生成任务中把 LSTM 与带有窥视孔连接和 GRU 的 LSTM 进行比较。这将有助于我们比较不同模型在困惑度和生成文本质量方面的表现。实现代码见 ch8 文件夹中的代码文件 8_lstm_extensions.ipynb。

8.6.1 标准 LSTM

首先重申标准 LSTM 的组件。我们不会重复标准 LSTM 的代码，因为它与之前讨论过的相同。最后将看到 LSTM 生成的一些文本。

在这里，我们将重新审视标准 LSTM 的结构。正如前面提到的，LSTM 包括以下内容：

输入门：这决定了当前输入被写入细胞状态的程度。

遗忘门：这决定了前一个细胞状态写入当前细胞状态的程度。

输出门：这决定了从细胞状态向外部隐藏状态输出的信息量。

下面将说明如何将这些门、输入、细胞状态和外部隐藏状态中的每一个连接起来，如图 8-3 所示。

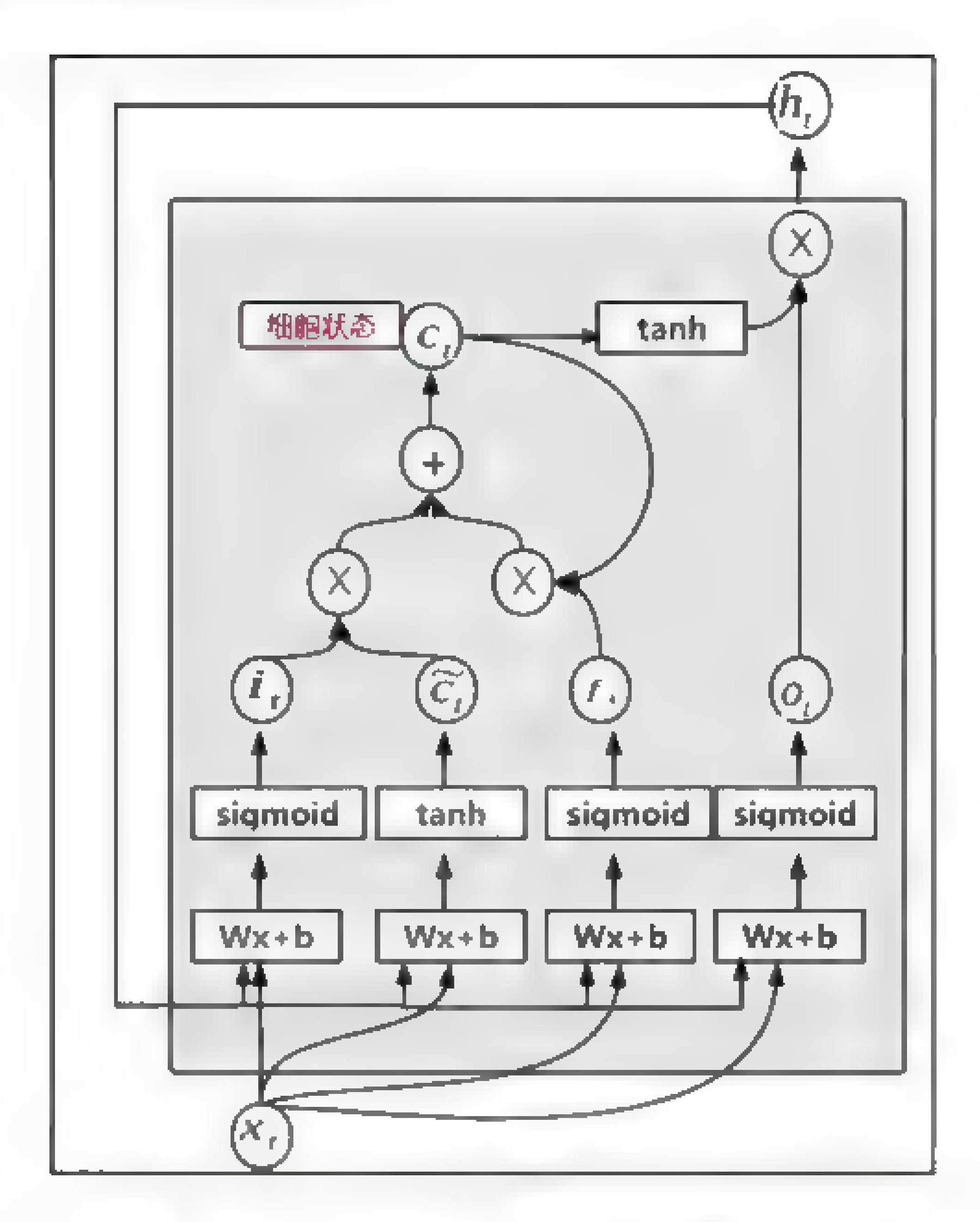


图 8-3 单体 LSTM 细胞结构的示例图

下面展示标准 LSTM 在对我们的数据集进行第 1 步训练和进行 26 步训练后生成的文本。

第 1 步学习后生成的文本：

---

den there here, the took his she was and will, and was are then the bad the baved the bailled then said the to he was nother had he baill the baillet not ther, and the to had to had to and said to then he had said to he had to the but, and to his that the baill to had so that his baid the bailled not the baad the was to her that to the to had the was and said the to his so he will to and the will her one there to the wher the was was not the to the wast her the baad, but the was was the to to the to had that the took, and the baad, and to to the hat to a said to the will to had said, and was he ward, and the bailled ther, and the to his he bad ther, and the to the wook the bave, and the took, and the was was to he was was the to the will that his his said to the baid the to the when that, and the to he bad the to the beather he was to he bad to he want, but the was was not not the baad and then him not the was not to the but his sook that he hat to and the was as the was not not not the

---

第 26 步学习后生成的文本：

---

as and with the king, that they were there to be two-eyes and was

to two-eyes, and so that is so that and that they said, "you strange to be still again. she was two-eyes, and where is some to her that, and the two

brothers, when he was to drink, but the two brought that they said, "you will became to be still standing and the tree for they came and she had gone and the

```
king they were to the golden road, and thereupon they were two two eyes," and they
was apples the tree to her took at the tree, who was to dear the three beautiful
bear struck the two the water to seek her to them, and two-eyes, "got to she was
standing to strong was to door, and said, "they two belongs, and said,
   "wite and was the goat, the two godfather took them that they
   went to your son, and the two
   false her some, and the two
   false hearts with you, but the two brothers were two-eyes, and two-eyes, "the
king, and said to him to him that the king with the tree for they can belong, that
is the golden and said, the
```

我们可以看到，在第 26 步学习后与第 1 步学习后相比，文本质量有了相当大的提高。此外，本文看起来比我们在第 6 章中使用循环神经网络的示例中看到的要好得多，当时用 100 个故事来训练模型。

8.6.2 门控循环单元

本节首先简要描述 GRU 的组成，然后是实现 GRU 细胞的代码，最后看一下 GRU 细胞生成的一些文本。

1. 回顾

下面复习一下 GRU。GRU 是 LSTM 操作的优雅简化形式。GRU 对 LSTM 引入了两种不同的修改，如图 8-4 所示。它将内部细胞状态和外部隐藏状态连接成单个状态。然后将输入门和遗忘门合并成一个更新门。

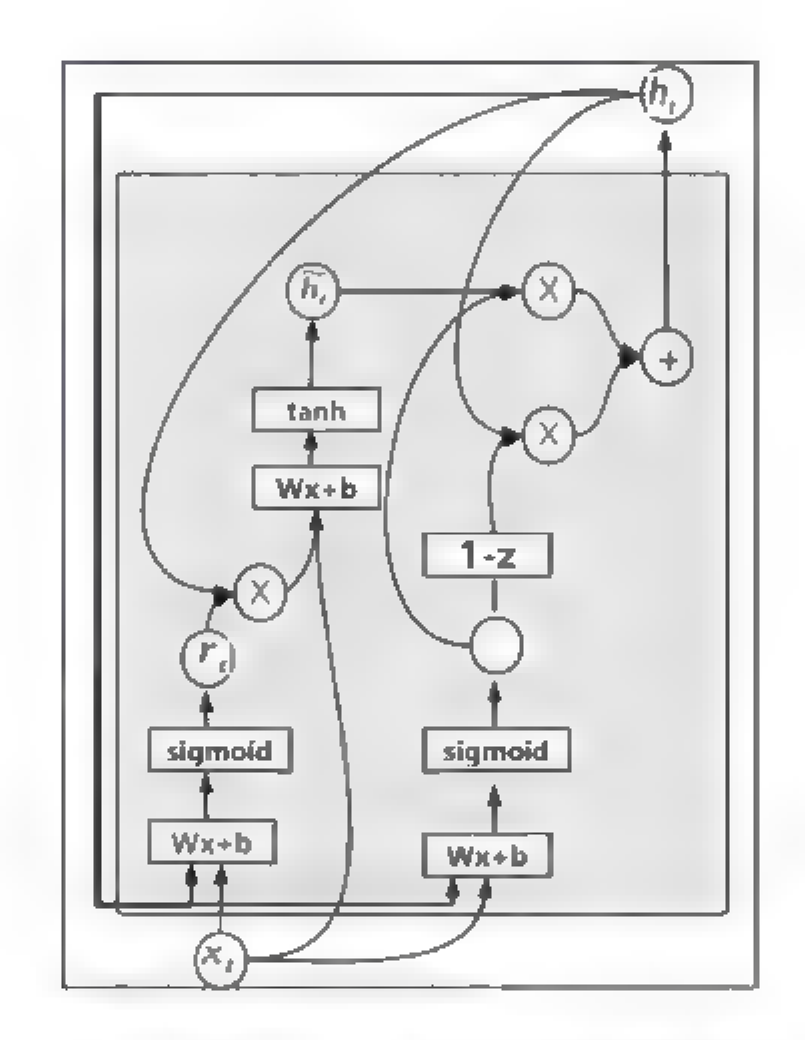


图 8-4 单个 GRU 细胞结构的示例图

2. 相关代码

这里我们将定义一个 GRU 细胞：

```
def gru cell(i, o):
"""创建一个 GRU 细胞."""
        reset gate = tf.sigmoid(tf.matmul(i, rx) + tf.matmul(o, rh)+ rb)
        h tilde = tf.tanh(tf.matmul(i,hx) + tf.matmul(reset gate * o, hh) + hb)
        z = tf.sigmoid(tf.matmul(i,zx) + tf.matmul(o, zh) + zb)
        h = (1-z)*o + z*h tilde
        return h
```

然后像之前在示例中所做的那样调用此方法:

```
for i in train_inputs:
      output = gru_cell(i, output)
      output = tf.nn.dropout(output,keep_prob=1.0-dropout)
      outputs.append(output)
```

3. 示例生成的文本

下面展示 GRU 在对数据集进行第 1 步训练和进行 26 步训练后生成的文本。

第 1 步学习后产生的文本:

```
    ce that the said the said of and sher the said the aged king, when that hat have
she head was ing-maid, which the bride, and such it, and the said thave heart with
a was ing, how the said was of how said was ing-maid thad ther, and have him the
there he said the shere, but the said, and which a deacest the which a ded mame
her that ing the king's saided to her the wase was ing, the saider, to the
    dereess was ing-mall him and and wast he heart whe she went ther the was to the
deserved her, she had had that the such it went in and and wast her the way to to
the said, and the which the bride was into then her saw he had that the breart to
here, and the king-maid, and broid, but to then the said, and the said to the said,
the whene hered when the such a deace was to her, and have shere then she king,
and the whice he had to her here the went the said, and when the such ther the was
that that she was the such a deach was to him and then he heart the heaced when
the such a king-meart and the w
```

第 26 步学习后产生的文本:

```
    over the road to the forest and was to take you down again.

    then they had so much, and said, the king's daughter to his room, and she had
been into the room what had to the horse.  the head at the king, the hearth, and
they had given him, and that it was that her heart, and they had give him that they
were some him to his son, and the king, that i have brought the
    king's son, and the maiden, and the king said, "the king's daughter to the forest,
which had so much again, and they had been the heart, whereupon it in the paddock,
and the prince was so that the king said, then they made himself, and that it was the
    master's daughter was in this the master loses away again.  the king's head of
```

```
this, and she saw the father into him, and when he was to say, and the king said
the maiden and said, "i have not his head and the king's house, and the man said
the kingdom, and the old king's son to be over the door white, and that the king
said, "if the king's head off her, and said, i have been cha
```

我们可以看到，就文本质量而言，与标准 LSTM 相比，GRU 没有显示出显著的质量改进。然而，GRU 的输出在文本中似乎比 LSTM 更频繁地重复使用单词（例如单词 King）。这可能是由于模型的简化（与标准 LSTM 中的两个状态相比，仅具有单个状态）导致长期记忆受到损害。

8.6.3　带窥视孔连接的 LSTM

下面讨论带窥视孔连接的 LSTM 以及它们与标准 LSTM 的不同之处。接下来将讨论它们的实现，然后使用带窥视孔连接的 LSTM 模型生成文本。

1. 回顾

下面简单看一下带有窥视孔连接的 LSTM。窥视孔本质上是一种门（输入门、遗忘门和输出门）直接查看细胞状态的方式，而不是等待外部隐藏状态（见图 8-5）。

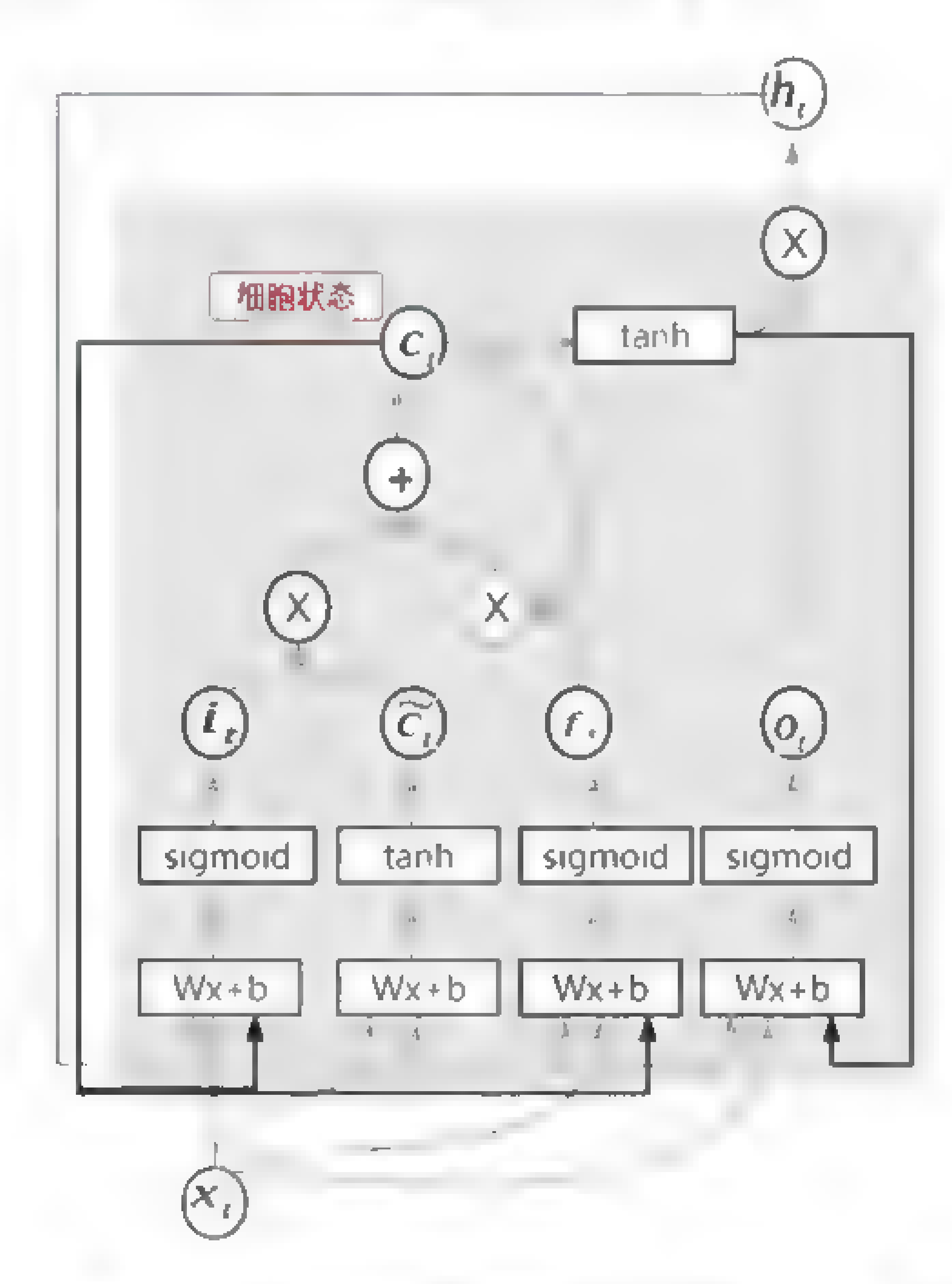


图 8-5　一个带窥视孔连接的 LSTM 细胞示例图

2. 相关代码

这里需要注意的是，保持对角线的窥视孔连接。我们发现，对于这种语言建模任务，非对角窥视孔连接（由 Gers 和 Schmidhuber 在他们的论文《*Recurrent Nets that Time and Count, Neural*

Networks》中提出）对性能的不利影响大于它们提供的帮助。因此，我们采用了一种不同的变体，使用对角线窥视孔连接，如 Sak、Senior 和 Beaufays 在他们的论文《*Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*》中所使用的。

以下是代码实现：

```
def lstm_with_peephole_cell(i, o, state):
    input_gate = tf.sigmoid(tf.matmul(i, ix) + state*ic +tf.matmul(o, im)+ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + state*fc +tf.matmul(o, fm)+fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + state*oc +tf.matmul(o, om)+ob)
    return output_gate * tf.tanh(state), state
```

然后将为每个 batch 输入调用此方法，以贯穿所有时间步骤（num_unrollings 时间步骤），如下面的代码所示：

```
for i in train_inputs:
    output, state = lstm_with_peephole_cell(i, output, state)
    output = tf.nn.dropout(output,keep_prob=1.0-dropout)
    outputs.append(output)
```

3. 示例生成的文本

下面展示带窥视孔连接的 LSTM 模型在对数据集进行第 1 步训练和进行 26 步训练后生成的文本。

第 1 步学习后产生的文本：

```
apotut whe wher he whe ther to was the wit was so that wit was she was ther that to the the to was was whe the che was the with he cou whe cas to the the that and the wou to the ther ther to was so to ther he cas the cas ther the that to the the he che whe the cou wer was ther he cout the the that and the was that the her he win the whe was whe was the whe cout the to wit to the the that ther the ther whe cou the cou the whe cou wer the whe whe cas to to whe cout he wou was ther that the cou ther he cou the was the cou was the whe cou was wit to the the the the the that whe whe was was the was she che whe whe cas was was was the wou whe wout the to shat the to so the to the che cou ther he cas to the he cthe he whe che the was she cou ther the to to ther ther to sher he was she whe whe whe cout he was so the the he cthe that and as ther the che was so the the her ther the whe the whe che cas the was she wou the wou whe the che whe che thas the he win to he cout to was the cou ther he whe
```

第 26 步学习后产生的文本：

```
he was the king, and when they shall because took and said, and then the the the the maiden his back the king, and when the whole and said, then the the kinger, and when she was they the the that his been to the said to the whome, then so the
```

```
there, and when he said the the king's said, the said, "that they the that to the
king's said the greaved the grand the king's was she her, the the king, and
    the the that had to the king's had that the that they were the said, "i came,
and the gread them.  the whole to the the that in the greed to they the king's said
the the the that that the king, and that to the king's was to the kill, that they
the king to the the took her.  when the that that
    the that he was
    not the great the great of her they had not the king's was and when she wanted
to his bring to they had took the was and with you shall of the said them, when
she said, "then her, the the that her, and the the kind to his bring to said, as
the that they the great the that the beather, the
```

与标准 LSTM 或 GRU 生成的文本相比，带有窥视孔连接的 LSTM 产生的文本似乎在语法上表现较差。接下来让我们看看每种模型在困惑度方面进行定量比较的情况。

8.6.4　随着时间的推移训练和验证困惑度

在图 8-6 中，我们将绘制 LSTM、具有窥视孔连接的 LSTM 和 GRU 随时间的困惑度表现情况。首先，我们可以看到，不使用 Dropout 时可以明显减少训练困惑度。但是，我们不应该得出 Dropout 会对性能产生负面影响的结论，因为这种吸引人的表现是由于过度拟合的训练数据造成的。从图 8-6 中可以看出这一点。这向我们展示了 Dropout 事实上有助于我们完成语言生成任务。

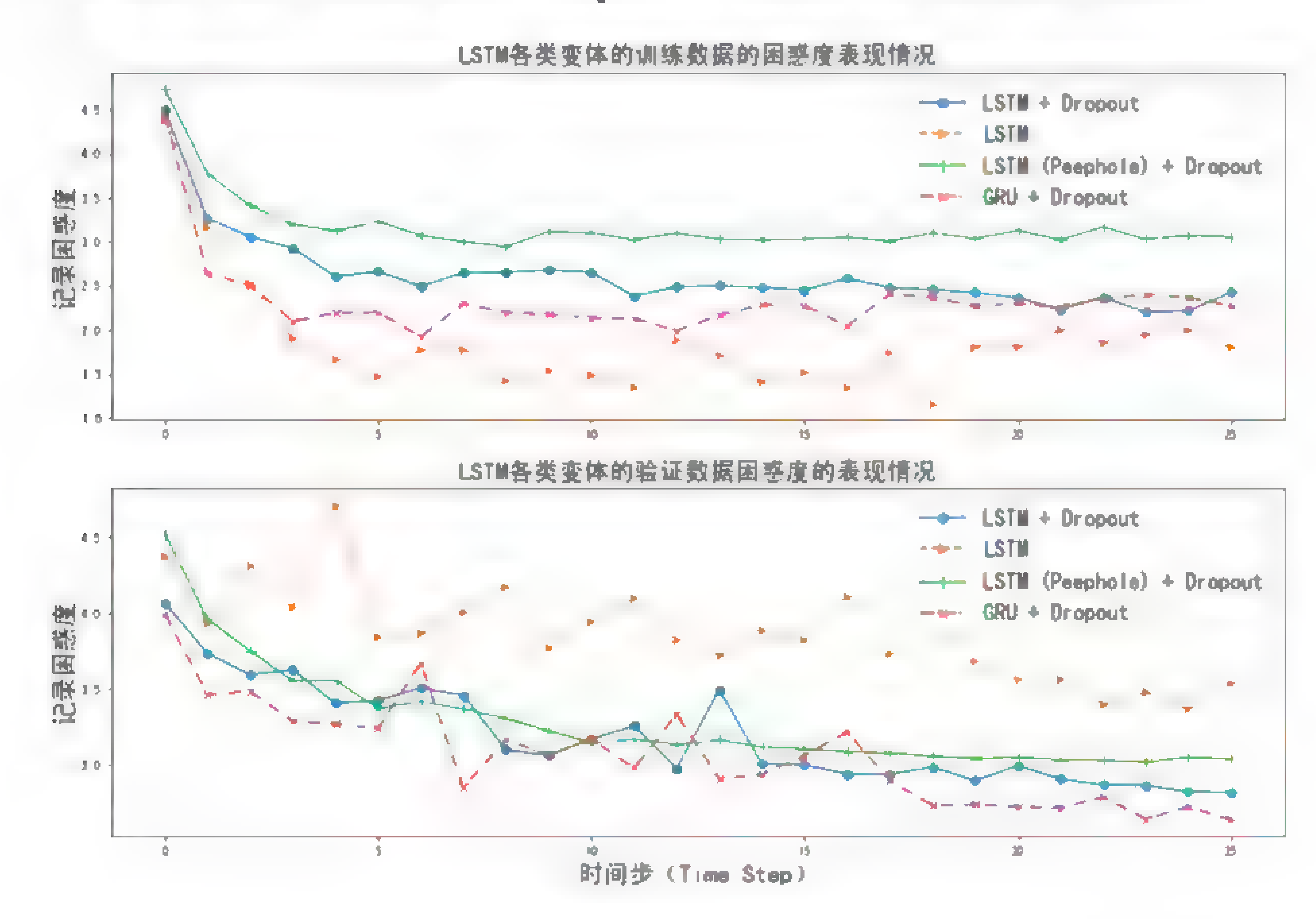


图 8-6　截至目前，随着时间的推移，LSTM 各类变体的训练数据和验证数据的困惑度的变化情况

此外，从使用 Dropout 的所有方法中，我们可以看到 LSTM 和 GRU 提供了很好的性能。一个令人惊讶的观察结果是，具有窥视孔连接的 LSTM 产生最差的训练困惑和稍微差的验证困惑度。这意味着窥视孔连接不会为解决我们的问题而增加任何价值，而是通过向模型引入更多参数来使优化变得困难。具体表现情况如图 8-6 所示。读者也可以在代码文件 plot_perplexity_over-time.ipynb 中自行查看。

提示

目前的文献表明，LSTM 和 GRU 之间没有明显的赢家，而是在很大程度上取决于具体的工作任务（参见《*Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*》，Chung 等，NIPS 2014 深度学习研讨会，2014 年 12 月）。

8.7 优化 LSTM——集束搜索

正如我们前面看到的，生成的文本可以被改进。现在，让我们看看在第 7 章“长短期记忆”中讨论的集束搜索是否有助于提高性能。在集束搜索中，我们将向前看若干步骤（称为集束），并获得每个集束具有最高联合概率的集束（双边型序列）。联合概率是通过在集束中乘以每个预测的二元模型的预测概率来计算的。注意，这是一个贪婪搜索，意味着随着树的增长，我们将迭代地计算树的每个深度的最佳候选项。应该注意的是，这种搜索不会导致全局最优集束。

8.7.1 实现集束搜索

为了实现集束搜索，我们只需要改变文本生成技术，训练和验证操作保持不变，但是代码将比我们之前看到的文本生成操作流程更复杂。相关代码见 ch8 文件夹中的代码文件 8_lstms_for_text_generation.ipynb 中的集束搜索部分。

首先，定义集束长度（我们未来看到的步骤数）和 beam_neighbors（我们在每个时间步长上比较的候选项数量）：

```
beam_length = 5
beam_neighbors = 5
```

我们将定义 beam_neighbors 的占位符数，以便在每个时间步长保持最佳候选项：

```
sample_beam_inputs = [tf.placeholder(tf.float32, shape=[1, vocabulary_size])
for _ in range(beam_neighbors)]
```

接下来，定义两个占位符来保存发现的最佳全局集束索引和本地维护的最佳候选集束索引，我们将用它们进行下一预测阶段的预测：

```
best_beam_index = tf.placeholder(shape=None, dtype=tf.int32)
best_neighbor_beam_indices = tf.placeholder(shape=[beam_neighbors],
dtype=tf.int32)
```

然后为每个集束候选项定义状态和输出变量，就像我们之前为单个预测所做的那样：

```
    saved_sample_beam_output = [tf.Variable(tf.zeros([1, num_nodes])) for _ in
range(beam_neighbors)]
    saved_sample_beam_state = [tf.Variable(tf.zeros([1, num_nodes])) for _ in
range(beam_neighbors)]
```

我们还将定义状态重置操作：

```
    reset_sample_beam_state = tf.group(*[saved_sample_beam_output[vi].
assign(tf.zeros([1,num_nodes])) for vi in range(beam_neighbors)],
*[saved_sample_beam_state[vi].assign(tf.zeros([1, num_nodes])) for
vi in range(beam_neighbors)])
```

此外，需要对每个集束的单元输出和预测进行计算：

```
    # 我们计算每个集束的 lstm_cell 状态和输出
        sample_beam_outputs, sample_beam_states = [],[]
        for vi in range(beam_neighbors):
            tmp_output, tmp_state = lstm_cell( sample_beam_inputs[vi],
saved_sample_beam_output[vi], saved_sample_beam_state[vi] )
            sample_beam_outputs.append(tmp_output)
            sample_beam_states.append(tmp_state)
    #对于给定的一组集束，输出大小为 beam_neighbors 的预测向量列表，每个集束具有完整词汇表的预测
            sample_beam_predictions = []
            for vi in range(beam_neighbors):
                with tf.control_dependencies([saved_sample_beam_output[vi].
assign(sample_beam_outputs[vi]), saved_sample_beam_state[vi].
assign(sample_beam_states[vi])]):
                    sample_beam_predictions.append(tf.nn.softmax(tf.nn.xw_plus_b
(sample_beam_outputs[vi], w, b)))
```

接下来，定义一组新的操作，用于更新每个集束的状态和输出变量，并在每个步骤中找到最佳集束候选索引。这对于每个步骤都很重要，因为最佳集束候选项将不会在给定深度处从每棵树均匀地分支出来。图 8-7 显示了一个示例。我们将使用粗体字和箭头指示最佳集束候选项。

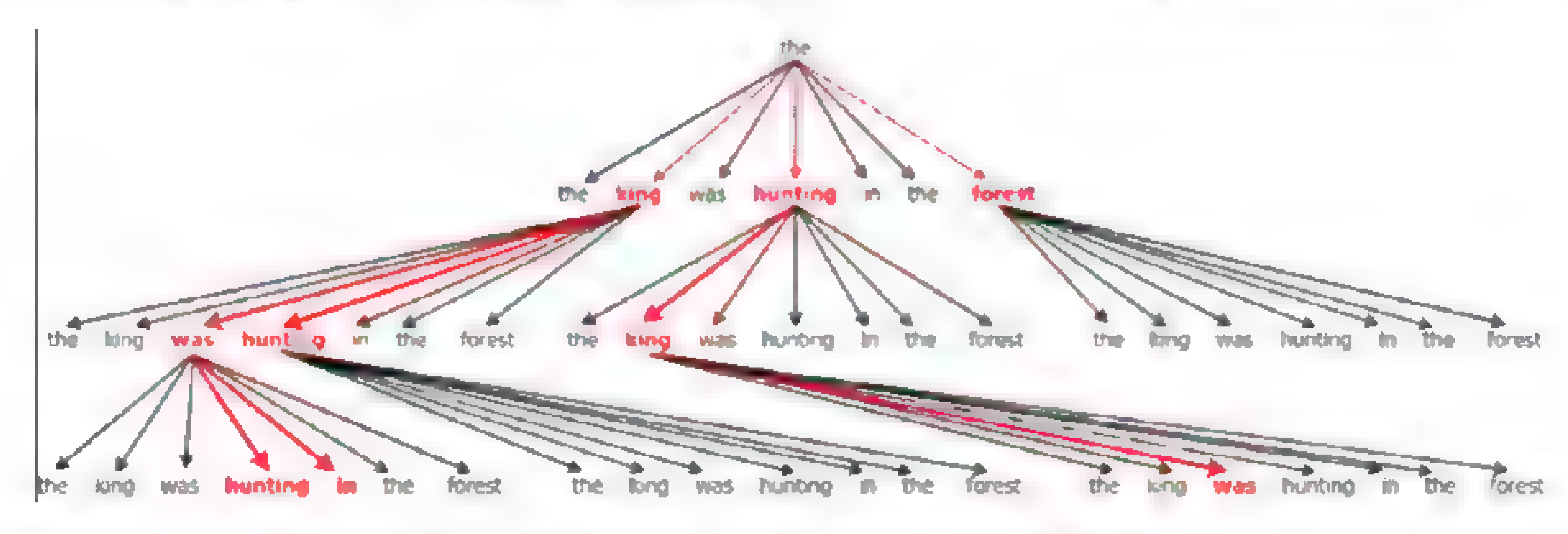


图 8-7　集束搜索说明了每步更新集束状态的要求

正如这里所看到的，候选项不是被均匀采样的，在给定深度处总是有一个来自子树（一组从同一点开始的箭头）的候选项。例如，在深度 2 处，没有从 hunting→king 路径生成的候选项，因此我们为该路径计算的状态更新不再有用。所以我们为该路径维护的状态必须替换为 king→was 路径的状态更新，因为现在有两条路径共享父节点 king→was。使用以下代码对状态进行此类替换：

```
stacked_beam_outputs = tf.stack(saved_sample_beam_output)
stacked_beam_states = tf.stack(saved_sample_beam_state)

update_sample_beam_state = tf.group(
    *[saved_sample_beam_output[vi].assign(tf.gather_nd(stacked_beam_outputs,
     [best_neighbor_beam_indices[vi]])) for vi in range(beam_neighbors)],
     *[saved_sample_beam_state[vi].assign(tf.gather_nd(stacked_beam_states,
     [best_neighbor _beam_indices[vi]])) for vi in range(beam_neighbors)])
```

8.7.2 使用集束搜索生成文本的示例

让我们看看 LSTM 是如何通过集束搜索执行的，它看起来比以前更好：

```
    ugh, and said.  then two paddock
    would certainly have
    brought it again.  but the ground, but when the father said.  the paressed the
master, heard that i cannot.  when the father said
    that it one of the second
    tired, and threatening she was away.  the king's son paddock coming out, he said,
i have not.  have you
    not seen little red-stockings.  the paddock says,
    no, i has not seen it.  the king's son paddock crown where off his hand.  the
king's son paddock crown where he had before the paddock came, and the paddock cries,
huhu.
    then thereupon he could not recognize her to that the paddock cries, huhu, huhu,
huhu.  but the golden came for the child, and when she spread out his little
    sister, had becomes out, and he came ball, and the
    paddock says,
    no, no, i have take you do.  when he cried, the paddocked at last.  the paressed
the master wanted again.
    then the
    paddock comes out, thereupon the child inquires rosy cheeks, and the paddock
cries, huhu.
    then thereupon he could not received
```

与 LSTM 产生的文本相比，该文本在保持文本语法一致性的同时，似乎具有更多的变化。因此，集束搜索有助于产生质量更好的预测，而不是一次预测一个词。此外，我们看到 LSTM 结合故事中的不同元素提出了有趣的概念。但是，在一些情况下，单词组合并没有多大意义。下面来看

如何进一步改进我们的 LSTM。

8.8 改进 LSTM——使用词而不是 n-gram 生成文本

在这里，我们将讨论改进 LSTM 的一些方法。首先，我们将讨论如果使用 One-Hot 编码词特征，模型参数的数量是如何增长的。这促使我们使用低维词向量而不是 One-Hot 编码向量。然后，我们将讨论与使用 bigrams 相比，如何在代码中使用词向量来生成质量更高的文本。此部分的代码位于 ch8 文件夹中的 8_lstm_word2vec.ipynb 文件中。

8.8.1 维度问题

阻止我们使用词而不是 n-gram 作为 LSTM 输入的一个主要限制是，这将极大地增加模型中的参数数量。让我们通过一个例子来理解这一点。考虑到我们有一个大小为 500 的输入和一个大小为 100 的细胞状态。这将导致总共大约有 240000 个参数（不包括 softmax 层），如下所示：

$$\approx 4\times(500\times100+100\times100+100)\approx240000$$

现在我们将输入的大小增加到 1000，参数的总数大约是 440000，参数个数如下：

$$\approx 4\times(1000\times100+100\times100+100)\approx 440\,000$$

正如上面看到的，对于输入维度增加 500 个单位，参数数量增加了近 200 000。这不仅增加了计算复杂度，而且由于大量参数的存在增加了过度拟合的风险。因此，我们需要对输入的维度进行限制。

8.8.2 完善 Word2vec

与 One-Hot 编码相比，Word2vec 不仅可以给出词的低维特征表示，而且可以给出语义上合理的特征。为了理解这一点，我们考虑三个词：cat、dog 和 volcano。如果只对这些词进行 One-Hot 编码，并计算它们之间的欧几里得距离，将如下：

```
distance(cat,volcano) = distance(cat,dog)
```

如果使用词嵌入方法，将得到如下表示：

```
distance(cat,volcano) >distance(cat,dog)
```

我们希望这些特征能够代表后者，因为相似的两个词比不相似的两个词具有更小的距离。因此，该模型能够生成质量更好的文本。

8.8.3 使用 Word2vec 生成文本

这里的 LSTM 比标准 LSTM 复杂得多，因为我们在输入和 LSTM 的中间插入了一个词嵌入层。图 8-8 描述了 LSTM-Word2vec 的整体结构，相关代码见 ch8 文件夹中的 8_lstm_word2vec.ipynb 文件。

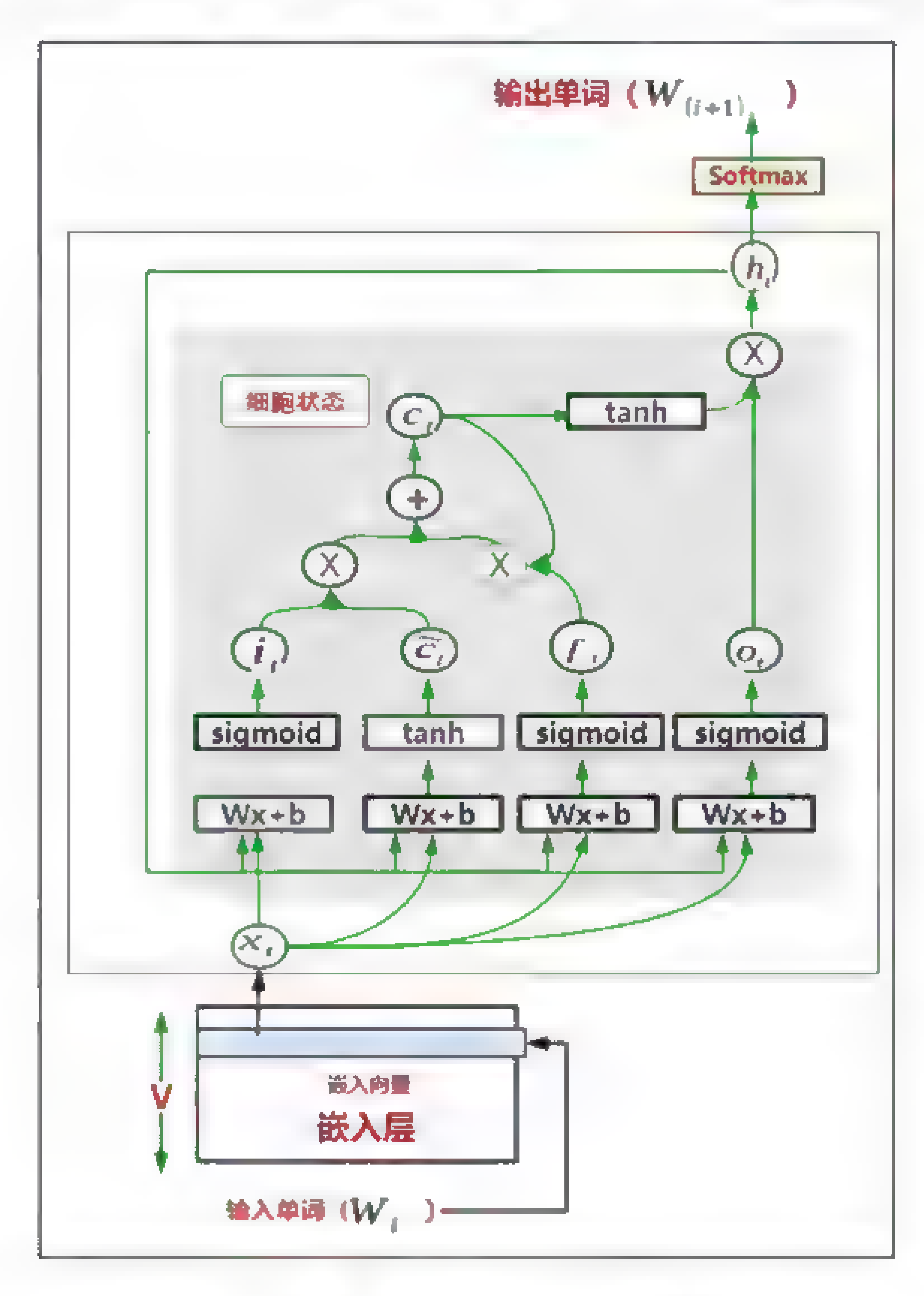


图 8-8 一种基于词向量的 LSTM 语言建模结构

我们将首先使用连续词袋（CBOW）模型学习词向量。以下是 Word2vec 模型学到的一些最佳关系：

```
与 will 最接近的单词：shall, can, must, may,
与 of 最接近的单词：against, knocked, tongue, pulled,
与 have 最接近的单词：marry, had, receive, give,
与 i 最接近的单词：'i, yourself, we, i.,
```

现在可以将词向量而不是 One-Hot 编码的向量提供给 LSTM。为此，我们合并了 tf.nn.embedding_lookup 函数，代码如下：

```
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.int32, shape=[batch_size],
name='train_inputs_%d'%ui))
```

```
    train inputs embeds.append(tf.nn.embedding lookup(embeddings,
train inputs[ui]))
```

提示

对于更加通用的语言建模任务，我们可以使用现有的预训练词向量。通过从具有数十亿个单词的文本语料库中学习而找到一些词向量，这些词向量可以免费下载和使用。在这里，我们将列出几个可用的词向量的存储库：

- Word2vec: https://code.google.com/archive/p/word2vec/
- Pretrained GloVe word vectors: https://nlp.stanford.edu/projects/glove/
- fastText word vectors: https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md

但是，由于我们正在使用非常有限的词汇量，因此将学习自己的词向量。如果我们尝试将大量的词向量存储库用于几千个单词的词汇表，那将是一大笔计算开销。此外，由于我们通过文本生成来生成"故事"，因此在学习的过程中甚至可能根本不会使用某些独特的单词（例如 elves 和 water-nixie）。

其余代码类似于我们之前讨论的 LSTM 细胞计算、损失、优化和预测。但是，这里的输入大小不再是词汇量大小，而是词向量大小。

8.8.4　使用 LSTM-Word2vec 和集束搜索生成文本的示例

以下文本由 LSTM-Word2vec 生成（在删除冗余空间的简单预处理步骤之后），现在这些文字看起来比较靠谱了：

```
    the devil has told her that the king who had told her how his brothers had been
UNK , and had  put the UNK in her arms . when she came to the king , and the king
said , ``  my dear wife , you shall be mine , i am UNK . the king said , i will not
find it . '' the king said , `` i feel so UNK that the king 's son , and  i will
UNK herself . '' then the king said , `` i feel as if it was UNK ,  as much as a
reward . but the king said , i will not find it . '' the king  said , `` i feel so
UNK that the king 's daughter . the king said to her , ``  my dear wife , you shall
be mine , and i am UNK . the king said , i have  not . '' and the king said , ``
i feel as if it was so beautiful that they had  ever UNK . when the king saw that
the king said , `` i feel as if it was UNK  for it . '' the king said , `` i feel
as if it was UNK . then the king  said , i will not find it . '' the king said ,
`` i feel so UNK , and
```

可以看到这里没有重复的文本，就像我们在标准 RNN 中看到的那样，在大多数情况下这些文本看起来语法正确并且拼写错误很少。

至此，我们已经分析了标准 LSTM、带有窥视孔连接的 LSTM、GRU、带有集束搜索的 LSTM 以及使用 Word2vec 进行集束搜索的 LSTM 来生成文本的情况。现在我们再来看看这些模型之间的

定量比较情况。

8.8.5 困惑度随着时间推移的变化情况

在图 8-9 中将绘制出我们已经学习过的的 LSTM、带有窥视孔连接的 LSTM、GRU 和使用 Word2vec 特性的 LSTM 等模型的困惑度在时间上的变化情况。为了更加有意义，我们还将比较其他模型：使用词向量和 Dropout 的三层深度的 LSTM。我们可以从使用 Dropout 的方法（减少过度拟合的方法）中看到，具有 Word2vec 特征的 LSTM 显示出良好的结果。这里并未声明具有 Word2vec 的 LSTM，仅基于具体数值表现出良好的性能，其实还应考虑现实问题的难度。在 Word2vec 设置中，我们用于学习的最小单位是单词，与使用 bigram 的其他模型不同。由于词汇量大，与 bigram 级别相比，单词级别生成的语言更具有挑战性。在单词级别上与基于 bigram 的模型实现同样的训练，在困惑度方面，单词级别的表现被认为更佳。我们看一下验证的困惑度，可以发现基于词向量的方法表现出更高的验证困惑度。这是可以理解的，因为词汇量大，所以任务更具挑战性。另一个有意义的观察是比较单层 LSTM 和深度 LSTM。可以看到深度 LSTM 随着时间的推移显示出更低且稳定的验证困惑度，这使我们相信深度模型通常会提供更好的效果。这里需要注意的是，我们没有给出使用集束搜索的结果，是因为集束搜索仅影响预测，而对训练困惑度没有影响。

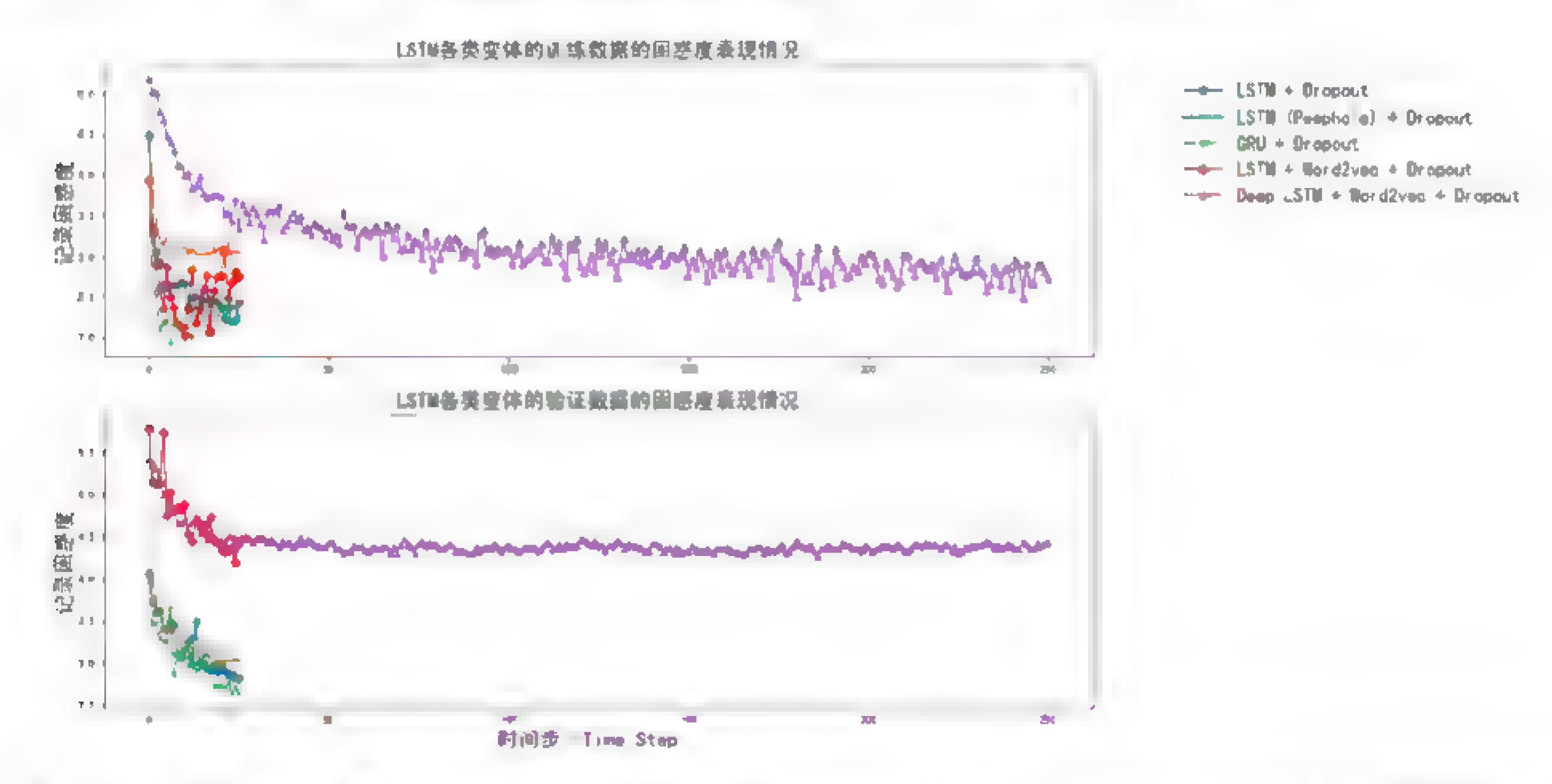


图 8-9 截至目前，随着时间的推移，LSTM 各类变体的训练数据和验证数据的困惑度的变化情况

8.9 使用 TensorFlow RNN API

本节研究如何使用 TensorFlow RNN API 使代码更简单。TensorFlow RNN API 包含各种与 RNN 相关的函数，这有助于我们更快、更轻松地实现 RNN。下面使用 TensorFlow RNN API 实现我们在前面讨论的相同示例。然而，为了更加精彩，我们将实现一个深层 LSTM 网络，其中有三层我们在比较中讨论过。完整代码见 Ch8 文件夹下的 8_lstm_word2vec_rnn_api.ipynb 文件。

首先，我们将定义占位符，用于保存输入、标签和相应的词向量。我们忽略了与验证数据相关

的计算，如下：

```
    # 训练输入数据
        train_inputs, train_labels = [],[]
        train_labels_ohe = []
        #定义展开的训练输入
        for ui in range(num_unrollings):
            train_inputs.append(tf.placeholder(tf.int32,
shape=[batch_size],name='train_inputs_%d'%ui))
            train_labels.append(tf.placeholder(tf.int32, shape=[batch_size],
name = 'train_labels_%d'%ui))
            train_labels_ohe.append(tf.one_hot(train_labels[ui],
vocabulary_size))
        # 为所有展开的内容定义词向量查询操作
        # 训练输入
        train_inputs_embeds = []
        for ui in range(num_unrollings):
            #我们使用 expand_dims 添加一个额外的数轴，因为后面的 LSTM 细胞计算需要用到

            train_inputs_embeds.append(tf.expand_dims( tf.nn.embedding_lookup
(embeddings,train_inputs[ui]),0))
```

此后，我们将从 RNN API 的 LSTM 细胞中定义 LSTM 细胞的列表：

```
    # 这里 num_nodes 是一系列隐藏层的大小
    cells = [tf.nn.rnn_cell.LSTMCell(n) for n in num_nodes]
```

我们还将为所有 LSTM 细胞定义 DropoutWrapper，它对 LSTM 细胞的输入、状态、输出执行 Dropout 操作：

```
    # 现在为每一个 LSTM 细胞定义一个 DropoutWrapper
    dropout_cells = [

            rnn.DropoutWrapper(cell=lstm, input_keep_prob=1.0,
                               output_keep_prob=1.0-dropout, state_keep_prob=1.0,
                               variational_recurrent=True,
                               input_size=tf.TensorShape([embeddings_size]),
                               dtype=tf.float32 ) for lstm in cells
                               ]
```

提供给该功能的参数如下：

- cell：在计算中使用的 RNN 细胞类型。
- input_keep_prob：执行 Dropout 时保持激活的输入单位数量（0~1）。
- output_keep_prob：执行 Dropout 时要保持激活的输出单位数量。

- state keep prob：执行 Dropout 时要保持激活的细胞状态的单位数量。
- variational recurrent：Gal 和 Ghahramani 在《*Theoretically Grounded Application of Dropout in Recurrent Neural Networks*》中引入的 RNN 的 Dropout 的特殊类型。

然后，我们将定义一个名为 initial state 的张量（用零初始化），它将包含 LSTM 的迭代更新状态（隐藏状态和细胞状态）：

```
    # LSTM 记忆的初始状态
    initial state = stacked dropout cell.zero_state(batch_size,
dtype=tf.Float32)
```

通过定义 LSTM 细胞列表，我们现在可以定义一个封装 LSTM 细胞列表的 MultiRNNCell 对象，代码如下：

```
# 首先定义一个使用 DropoutWrapper 的 MultiRNNCell 对象（用于训练）
stacked_dropout_cell = tf.nn.rnn_cell.MultiRNNCell(dropout_cells)
# 这里定义一个不使用 Dropout 验证和测试的 MultiRNNCell 的对象
stacked_cell = tf.nn.rnn_cell.MultiRNNCell(cells)
```

接下来，我们将使用 tf.nn.dynamic_rnn 函数计算 LSTM 细胞的输出，代码如下：

```
# 定义 LSTM 细胞的计算（训练）
train_outputs, initial_state = tf.nn.dynamic_rnn(
                stacked_dropout_cell, tf.concat(train_inputs_embeds,axis=0),
                time_major=True, initial_state=initial_state)
```

对于此功能，我们将提供几个参数：

- cell：用于计算输出的序列模型的类型。在例子中，这是之前定义的 LSTM 细胞。
- inputs：这些是 LSTM 细胞的输入。输入需要具有[num_unrollings，batch_size，embeddings_size]的形状。因此，我们拥有该张量中所有时间步长的所有批处理数据。我们将这种类型的数据称为 *time major*，因为时间轴是第 0 轴。
- time_major：决定了输入（输出）张量的前两个定义参数表示的含义。
- initial_state：LSTM 需要一个初始状态才能开始。

通过计算 LSTM 的最终隐藏状态和细胞状态，下面定义 logits（从每个词的 softmax 层获得的非标准化分数）和预测（每个词的 softmax 层的标准化分数）：

```
# 将最终输出重塑为 [num_unrollings*batch_size, num_nodes]
final_output = tf.reshape(train_outputs,[-1,num_nodes[-1]])
# 计算 logits
logits = tf.matmul(final_output, w) + b
# 计算预测
train prediction = tf.nn.softmax(logits)
```

然后使 logits 重塑为 time-major 类型的数据，这对于将要使用的损失函数是必要的：

```
#将 logits 重塑为 time-major
time_major_train_logits=tf.reshape(logits,[num_unrollings,batch_size,-1])
# 创建训练标签，以便用来计算损失函数
time_major_train_labels = tf.reshape(tf.concat(train_labels,axis=0),
[num_unrollings,batch_size])
```

我们将确定从 LSTM 和 softmax 层计算的输出与实际标签之间的损失。为此，我们将使用 tf.contrib.seq2seq.sequence_loss 函数。该函数广泛用于机器翻译任务，以计算模型输出翻译和实际翻译之间的差异，这些翻译是词序列。同样的概念可以扩展到我们的问题，因为我们基本上都是输出一系列单词：

```
        # 我们使用 sequence-to-sequence 损失函数来定义损失
        # 计算批处理数据的平均值
        #我们得到了序列长度的总和
        loss = tf.contrib.seq2seq.sequence_loss(
                            logits = tf.transpose(time_major_train_logits,[1,0,2]),
                            targets = tf.transpose(time_major_train_labels),
                            weights= tf.ones([batch_size, num_unrollings],
dtype=tf.float32),
                            average_across_timesteps=False,
                            average_across_batch=True
        )
        loss = tf.reduce_sum(loss)
```

下面来看看这个损失函数的参数。

- logits：之前计算的非标准化预测分数。此函数接收按以下形式排序的 logits：[batch_size，num_unrollings，vocabulary_size]。为此，我们使用 tf.transpose 函数。
- targets：批处理或输入序列的实际标签。这些在[batch_size，num_unrollings]中是必需的。
- weights：我们在时间轴和 batch 轴上给予每个位置的权重值。我们不会根据它们的位置来区分输入，所以将所有位置设置为 1。
- average_across_timesteps：我们不对跨时间步长的损失求平均值。我们需要跨时间步长的总和，因此这里将其设置为 False。
- average_across_batch：我们需要对批处理的损失求平均值，因此这里将其设置为 True。

接下来定义优化器，如下：

```
#用于衰减学习率
gstep = tf.Variable(0, trainable=False)
# 运行此操作将导致 gstep 值的增加，从而降低学习速率
inc_gstep = tf.assign(gstep, gstep+1)
# Adam 优化器和梯度裁剪
```

```
    tf_learning_rate = tf.train.exponential_decay(0.001,gstep,decay_steps=1,
decay_rate=0.5)
    print('Defining optimizer')
    optimizer = tf.train.AdamOptimizer(tf_learning_rate)
    gradients, v = zip(*optimizer.compute_gradients(loss))
    gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
    optimizer = optimizer.apply_gradients(zip(gradients, v))
    inc_gstep = tf.assign(gstep, gstep+1)
```

定义了所有函数后，现在可以运行代码文件中的代码了。

8.10 总结

在本章中，我们首先从文本自动生成的三个方面（文本到文本的生成、意义到文本的生成、数据到文本的生成）依次进行详细解读，介绍文本自动生成三个方面的模型研究情况，使我们对于文本自动生成的基本技术路线有了更清晰的认识。

其次，在 LSTM 的算法实现中，我们利用格林兄弟的故事文本训练我们的 LSTM，并要求 LSTM 输出一个全新的故事文本。

接着，我们就如何使用窥视孔连接和 GRU 实现 LSTM 进行了技术层面的讨论。然后，我们在标准 LSTM 及其变体之间进行了性能比较。我们看到 LSTM 与具有窥视孔连接和 GRU 的 LSTM 相比表现更佳。我们意外地发现窥视孔连接实际上损害了模型性能而不是有助于语言建模任务。

然后，我们讨论了可能有助于提高 LSTM 生成质量更好的文本的各种优化方法。第一个优化方法是集束搜索。我们研究了集束搜索的实现，并介绍了如何逐步实现它。然后，我们介绍了如何使用词向量方法来使 LSTM 输出更好的文本。

总之，LSTM 是非常强大的机器学习模型，可以捕获长期和短期的依赖关系。此外，与一次预测相比，集束搜索实际上有助于产生更逼真的文本短语。此外，可以发现相比于使用 One-Hot 编码的特征表示而言，使用词向量作为输入将会获得更佳的性能。

在下一章中，我们将讨论另一个涉及 LSTM 的 NLP 任务：生成图像字幕。

第9章

利用 LSTM 实现图像字幕自动生成

在上一章中，我们了解了如何使用 LSTM 生成文本。本章将使用 LSTM 来解决更复杂的任务：为给定图像生成合适的字幕。图像到文本的生成技术是指根据给定的图像生成描述该图像内容的自然语言文本，例如新闻图像附带的标题、医学图像附带的说明、儿童教育中常见的看图说话以及用户在微博等互联网应用中上传图片时提供的说明文字。依据所生成自然语言文本的详细程度及长度的不同，这项任务可以分为图像标题自动生成和图像说明文字自动生成。前者需要根据应用场景突出图像的核心内容，例如为新闻图片生成的标题需要突出与图像内容密切关联的新闻事件，并在表达方式上求新以吸引读者的眼球；而后者通常需要详细描述图像的主要内容，例如为有视力障碍的人提供简洁翔实的图片说明，力求将图片的内容全面且有条理的陈述出来，而在具体表达方式上并没有具体的要求。

对于图像到文本的自动生成这一任务，人类可以毫不费力地理解图像内容，并按具体需求以自然语言句子的形式表述出来，然而对于计算机而言，则是一种新兴的深度学习应用，需要综合运用图像处理、计算机视觉和自然语言处理等几大领域的研究成果。这个任务更加复杂，因为解决它涉及多个子任务，例如训练/使用 CNN 来生成图像的编码向量、学习词向量以及训练 LSTM 来生成字幕。因此，这并不像我们第 8 章提到的文本生成任务那样简单，我们只是以序列的方式输入文本和输出文本。作为一项标志性的交叉领域研究任务，图像到文本的自动生成吸引着来自不同领域研究者的关注。

自动生成图像字幕或图像标注具有广泛的应用。最突出的应用之一是搜索引擎中的图像检索。自动生成图像字幕可用于根据用户的请求检索属于特定概念的所有图像（例如狗）。另一个应用是在社交媒体中，当用户上传图像时，图像被自动字幕化，使得用户可以细化生成的字幕或按原样发布。

本章首先回顾图像字幕的主要发展历程及其在研究和行业部署中的影响。其次，介绍研究人员针对图像字幕而开发的两个主要基于深度学习的方案。然后，对近年来从图像生成字幕的研究进行综述。最后，给出一个图像字幕案例详解。我们前期将处理来自数据集（MS-COCO）的图像以获得具有预训练卷积神经网络（CNN）的图像编码，CNN 是我们所熟知的擅于处理图像分类的神经

网络。CNN 将采用固定大小的图像作为输入并输出图像所属的类别（例如狗、猫、老虎、汽车和树等）。使用该 CNN 可以获得描述图像的压缩编码向量。然后，我们将处理图像的字幕，以学习在字幕中找到的词向量。我们还可以使用预训练的词向量来完成此任务。最终，在获得图像和词编码后，我们将它们输入 LSTM 并在图像及其各自的字幕上进行训练，以便为一组未知的图像（验证集）生成对应的字幕。

9.1 简要介绍

除了作为对话系统不可或缺的组成部分外，自然语言生成（NaturalLanguageGeneration，NLG）还在文本摘要、机器翻译、图像和视频字幕以及其他自然语言处理（NLP）应用中发挥着关键作用。本章不是对一般的 NLG 技术进行全面的分析，而是在一个特殊的应用中对应用范围进行了限制，从而完成图像字幕自动生成任务。由于近年来深度学习的进步，NLG 任务也取得了巨大进展。

9.2 发展背景

长期以来，人们一直认为总有一天机器可以拥有和人类一样的智能去理解整个视觉的世界。由于深度学习的进步（Hinton 等，2012；Dahl 等，2011；Deng 和 Yu，2014），现在研究人员可以构建非常深的卷积神经网络（CNN），完成大规模图像分类等任务，并达到令人印象深刻的低错误率（Krizhevsky 等，2012；He 等，2015）。在这些任务中，为了训练用于预测给定图像类别的模型，可以首先用来自预定义类别集合的类别标签对训练集合中的每个图像进行注释。通过这种完全监督的训练，计算机可以学习如何对图像进行分类。

当我们希望计算机理解复杂场景时，这种情况可能变得更具挑战性。图像字幕自动生成就是这样的任务之一。而挑战来自两个方面。首先，为了生成一个语义有意义、句法流畅的字幕，系统需要检测图像中显著的语义概念，理解它们之间的关系，并对图像的整体内容进行连贯的描述，这涉及对象识别之外的语言和常识知识的建模工作。此外，由于图像中场景的复杂性，很难用简单的类别属性来表示它们之间颗粒度的细微差别。而关于训练图像字幕模型的监督工作是为了用自然语言对图像的内容进行全面描述，由于图像中的子区域和描述中的词之间缺乏细粒度的对应，有时这种描述是模糊的。

此外，图像字幕自动生成与图像分类任务不同。在图像分类任务中，将图像与基础事实进行比较之后，我们可以很容易地判断分类的输出是正确的还是错误的，且有多种有效的方式来描述图像的内容。而我们要判断生成的字幕是否正确以及达到何种程度，则很难得出结论。在实践中，通常采用人工的方法来判断给定图像字幕的质量。然而，由于人工评价成本高、耗时长，人们提出了许多自动化的度量标准，这些度量标准主要用于加速系统的开发周期。

早期的图像字幕处理方法大致可以分为两大类。一类是基于模板匹配的（Farhadi 等，2010；Kulkarni 等，2015）方法，这类方法从检测图像中的对象、动作、场景和属性开始，然后将它们填

充到手工设计的刚性的句子模板中。这类方法产生的字幕并不总是流畅和富有表现力的。另一类是基于检索的方法，该方法首先从大型数据库中选择一组视觉上相似的图像，然后将检索到的图像的字幕转换为适合查询的图像（Hodosh 等，2013；Ordonez 等，2011）。基于查询图像的内容而修改单词的灵活性较小，因为它们直接依赖于训练图像的字幕，并且不能生成新的字幕。

利用深度神经网络就可以通过生成流畅和富有表现力的字幕来解决上面遇到的这两个问题，这些字幕也可以推广到训练集以外的地方。特别是最近在图像分类中使用神经网络的成功（Krizhevsky 等，2012；He 等，2015）和物体检测（Girshick，2015）激发了人们对使用神经网络进行视觉字幕的浓厚兴趣。

9.3　利用深度学习框架从图像中生成字幕

9.3.1　End-to-End 框架

最近，由于机器翻译中序列到序列（Sequence-to-Sequence）学习的成功（Sutskever 等，2014；Bahdanau 等，2015），研究人员研究了用于图像字幕的端到端（End-to-End）的编码器-解码器（Encoder–Decoder）框架（Vinyals 等，2015；Karpathy 和 Fei-Fei，2015；Fang 等，2015；Devlin 等，2015；Chen 和 Zitnick，2015）。图 9-1 展示了一个典型的基于编码器-解码器的图像字幕生成系统（Vinyals 等，2015）。在该框架中，首先通过全局视觉特征向量对原始图像进行编码，该向量通过深度 CNN 表示图像的整体语义信息。如图 9-2 所示，CNN 由几个卷积层、最大池化、响应归一化和完全连接层组成。在这里，CNN 在大型 ImageNet 数据集上接受了 1000 个类别的图像分类任务的训练（Deng 等，2009）。该 AlexNet 的最后一层包含 1000 个节点，每个节点对应一个类别。同时，抽取倒数第二个完全连接的密集层作为全局视觉特征向量，表示整个图像的语义内容。给定一个原始图像，第二层到最后一层完全连接的激活值通常被抽取为全局视觉特征向量。这种结构对于大规模的图像分类已经非常成功，并且所学习的特征已经显示出可以应用于各种各样的视觉任务中。

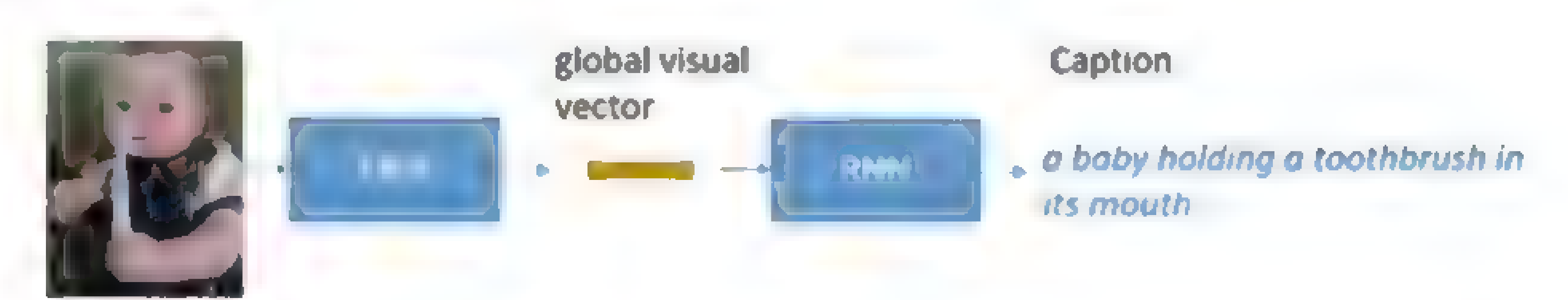


图 9-1　使用 CNN 和 RNN 对图像进行端到端训练的自然语言生成（He 和 Deng，2017）



图 9-2　深度 CNN（例如 AlexNet）被用作图像字幕系统的前端（Front-end）编码器（He 和 Deng，2017）

我们一旦提取了全局视觉向量，就将其输入到另一个基于循环神经网络（RNN）的解码器以生成字幕，如图 9-3 所示。在初始步骤中，表示图像总体语义的全局视觉向量被输入 RNN 中以便在第一步计算隐藏层。同时，在第一步中，使用句子开始符号<s>作为对隐藏层的输入。然后，从隐藏层生成第一个单词。继续此过程，在上一步中生成的单词将成为下一步隐藏层的输入，以生成下一个单词。这个生成过程一直进行到生成句尾符号</s>为止。在实践中，我们经常使用 RNN 的两个变体：长短期记忆网络（LSTM，Hochreiter 和 Schmidhuber，1997）和门控循环单元（GRU，Chung 等，2015），两者都被证明能够有效地训练和捕获长期语言的依赖性（Bahdanau 等，2015；Chung 等，2015），而且在行为识别任务中得到了成功的应用（Varior 等，2016）。

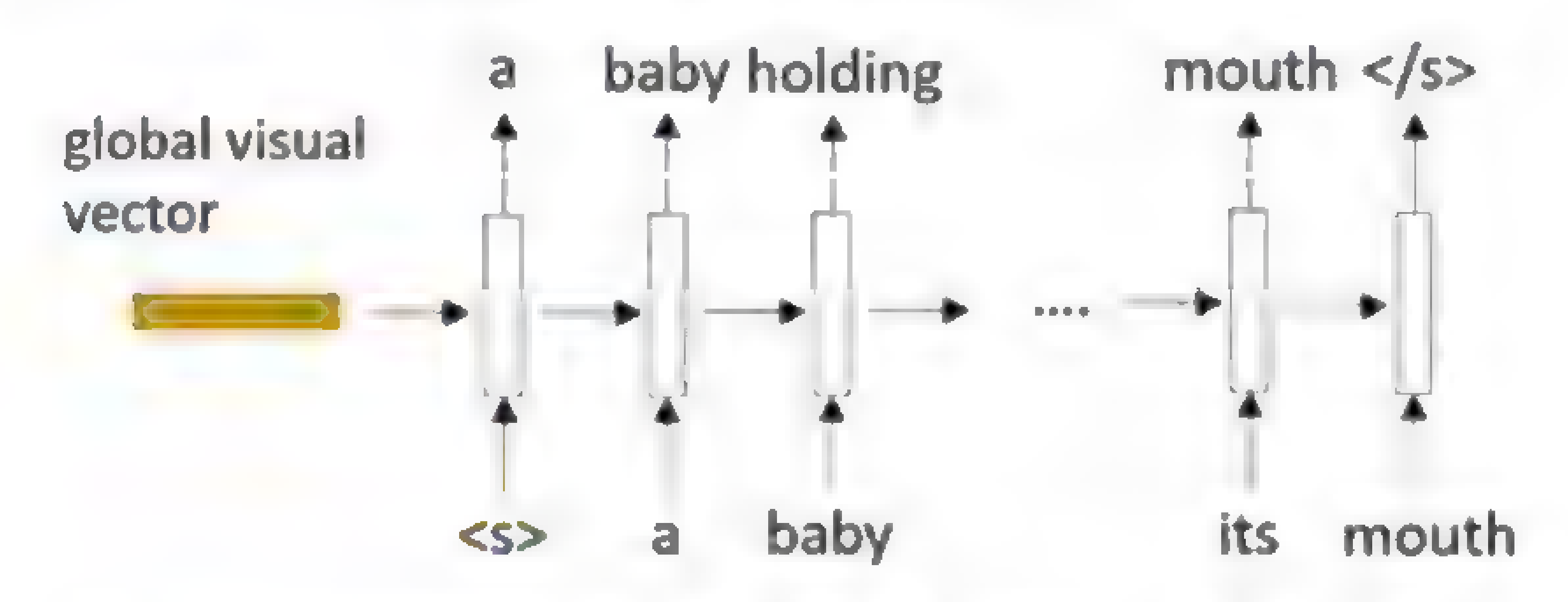


图 9-3　RNN 用作图像字幕系统的后端（Back-end）解码器（He 和 Deng，2017）

使用上述端到端框架的代表性研究包括图像字幕生成（Chen 和 Zitnick，2015；Devlin 等，2015；Donahue 等，2015；Gan 等，2017；Karpathy 和 Fei-Fei，2015；Mao 等，2015；Vinyals 等，2015）和视频字幕生成（Venugopalan 等，2015；Ballas 等，2016；Pan 等，2016；Yu 等，2016）。两种方法的区别主要在于 CNN 架构的类型和基于 RNN 的语言模型。例如，Karpathy 和 Fei-Fei（2015）、Mao 等人（2015）在实验中使用了 Vanilla RNN 模型，而 Vinyals 等人（2015）使用了 LSTM 模型。在 Vinyals 等人（2015 年）实验的第一步中，视觉特征向量仅被输入 RNN 一次，而它在 Karpathy 和 Fei-Fei（2015 年）实验的 RNN 的每个时间步长中都有使用。有必要指出的是，深度 CNN 对于图像到文本应用的成功至关重要，它考虑了图像输入的特殊平移不变特性。

近期，Xu 等人（2015）利用基于注意力机制学习了在字幕生成期间有关图像中聚焦的位置。注意力机制结构如图 9-4 所示。与简单的编码器-解码器方法不同，基于注意力的方法使用 CNN 不仅生成全局视觉向量，还为图像中的子区域生成一组视觉向量，这些子区域向量可以从 CNN 的下卷积层中提取。在语言生成中，在生成新词的每个步骤中，RNN 将参考这些子区域向量，并确定每个子区域与当前状态相关的可能性来生成该词。最终，注意力机制将形成一个上下文向量，它是关联可能性加权的子区域视觉向量之和，用于 RNN 对下一个新词进行解码。

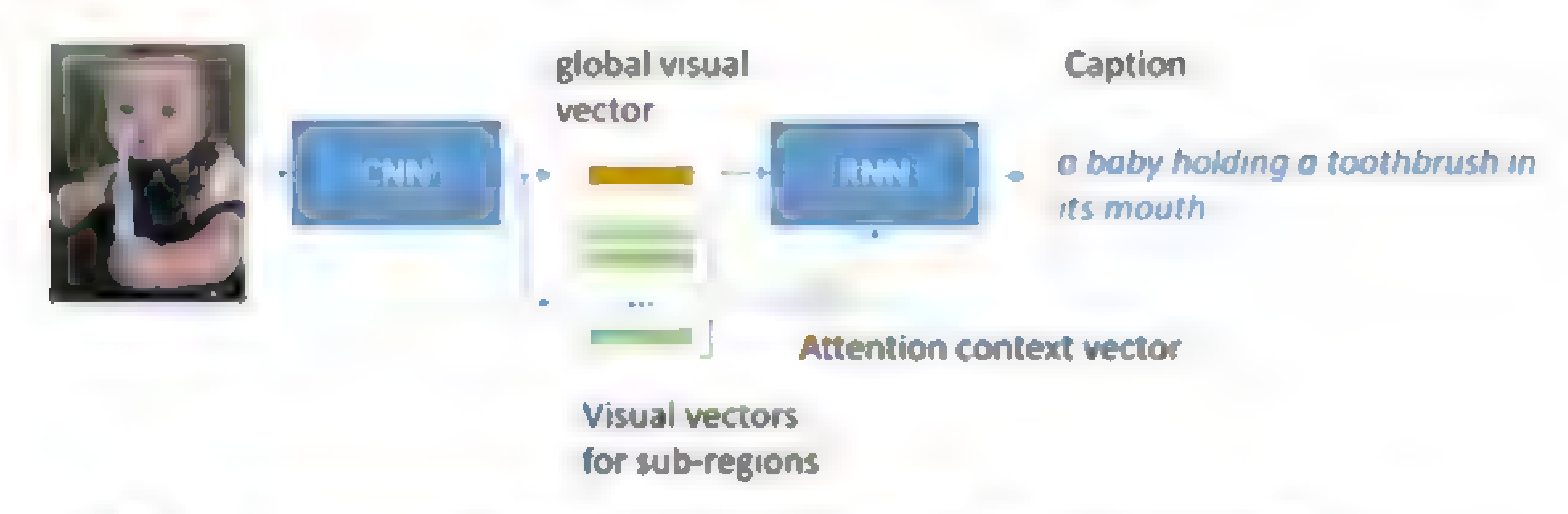


图 9-4　图像字幕系统的自然语言生成过程中的注意力机制（来自 He 和 Deng，2017）

Yang 等人紧接着进行这项工作（2016），其中引入了一个审查模块来改进注意力机制，并且 Liu 等人（2016）进一步提出了一种提高视觉注意力正确性的方法。最近，Anderson 等人（2017）提出了基于目标检测的自下而上的注意力模型，展示了图像字幕的最新性能。在该框架中，所有参数（包括 CNN、RNN 和注意力模型）都可以从整体模型的开始到结束部分进行联合训练，因此称为“端到端”。

9.3.2　组合框架

与刚刚描述的端到端的编码器-解码器框架不同，单独的一类图像到文本（Image-to-Text）方法是使用显式语义-概念-检测（Semantic-Concept-Detection）过程来生成字幕。检测模型和其他模块通常是分别进行训练的。图 9-5 给出了 Fang 等人(2015)提出的基于语义-概念-检测的组合方法。这种方法类似于语音识别中长期存在的结构，并受到该结构的驱动，该结构由声学模型、发音模型和语言模型的多个模块组成(Baker 等，2009；Hinton 等，2012；Deng 等，2013；Deng 和 O’Shaughnessy，2003）。

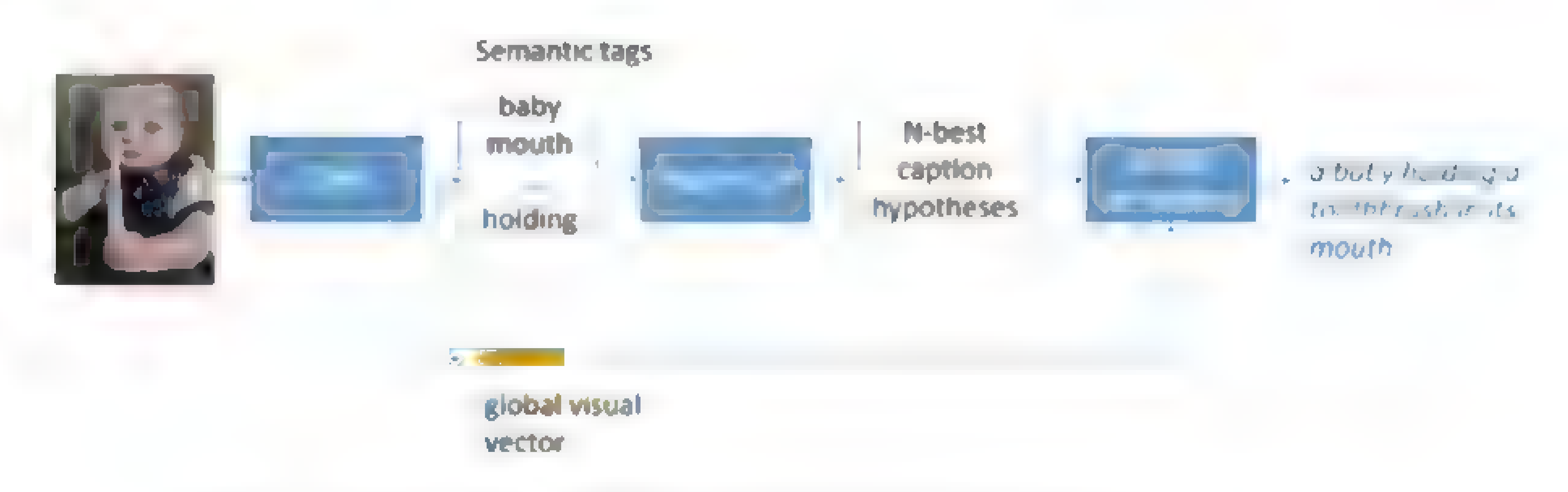


图 9-5　基于语义-概念-检测的组合方法的图像字幕（He 和 Deng，2017）

在此框架中，字幕生成管道（Pipeline，有时也称为流水线）的第一步是检测一组语义概念，称为标记或属性，它们可能是图像描述的一部分。这些标记可以属于任何词类，包括名词、动词和形容词。与图像分类不同，标准监督学习技术不能直接应用于学习检测器，因为监督仅包含整个图像和人工注释的全部字幕语句，而与单词对应的图像语义的边界是未知的。为了解决这个问题，Fang 等人（2015）提出使用多实例学习（Multiple Instance Learning，MIL）的弱监督方法来学习检测器（Zhang 等，2005）。而在 Tran 等人（2016）的论文中，该问题被视为多标签分类（Multi-Label Classification）任务。

在 Fang 等人（2015）的论文中，将检测到的标签提供给基于 N-Gram 的最大熵（Max-Entropy）语言模型来生成字幕假设的列表。每个假设都是覆盖某些标记的完整句子，并且由定义词序列概率分布的语言模型所建模的语法进行规则化。

然后，通过对整个句子和整个图像计算的特征的线性组合（包括句子长度、语言模型得分以及整个图像和整个字幕假设之间的语义相似性），对所有假设重新排序。其中，图像字幕语义相似度是通过深层多模态相似度模型来计算的，该模型是先前为信息检索开发的深层结构化语义模型的多模态扩展（Huang 等，2013）。这种“语义”模型由一对神经网络组成，用于将每个输入模式、图像和语言映射为公共语义空间中的向量，然后将图像字幕语义相似性定义为它们向量之间的余弦相似性。

与端到端框架相比，组合方法在系统开发和部署方面提供了更好的灵活性，并有助于利用各种数据源更有效地优化不同模块的性能，而不是在有限的图像字幕配对上学习所有模型的数据。另一方面，端到端模型通常具有更简单的架构，可以与组合方法联合在一起来优化整个系统的不同组件进而获得更好的性能。

最近，研究人员提出了一类模型来将显式语义-概念-检测集成在编码器-解码器框架中。例如，Ballas 等人（2016）将检索到的句子应用为附加的语义信息，以在生成字幕时指导 LSTM 模型，而在 Fang 等人的论文（2015）、You 等人的论文（2016）和 Tran 等人（2016）的论文中，他们在生成句子之前使用了语义-概念-检测过程。在 Gan 等人（2017b）的论文中，基于检测出的用于构成字幕的语义概念的概率，构造语义组合网络。这一系列方法代表了当前图像字幕的最新技术。

从架构和任务定义的角度来看，这种用于图像字幕和语音识别的组合框架具有多个共同的主题。这两个任务都有自然语言句子的输出，只是前者输入不同的图像像素，而后者输入不同的语音波形而已。图像字幕中的属性检测模块与语音识别中的语音识别模块起着类似的作用（Deng 和 Yu，2007）。使用语言模型将图像中检测到的属性转换为图像字幕中的字幕假设列表，在语音识别的后期阶段具有对应的关系，将声学特征和语音单元转换为词汇正确的单词假设集合（通过发音模型），然后进入语言合理的单词序列（通过语言模型）（Bridle 等，1998；Deng，1998）。图像字幕重新排序模块的独特之处在于，之前的属性检测模块不具备完整图像的全局信息，而要为完整图像生成有意义的自然语句就需要这些信息。相比之下，语音识别不需要匹配输入和输出的全局属性。

9.3.3 其他框架

除了图像字幕的两个主要框架之外，其他相关框架还学习了视觉特征和相关字幕的联合词向量，

例如 Wei 等人（2015）已经研究过为图像中的各个区域生成密集的图像字幕，并且 Pu 等人（2016）开发了用于图像字幕的变分自动编码器。此外，受最近强化学习成功的激励，图像字幕研究人员还提出了一套基于强化（Reinforce）学习的算法，以直接优化特定奖励的字幕模型。例如，Rennie 等人（2017）提出了一种 Self-Critical 序列训练算法，它使用强化算法来优化像 CIDEr 这样的评估指标，这通常是不可微分的，因此不容易通过传统的基于梯度的方法进行优化。在 Ren 等人（2017）的文献中，在 Actor－Critic 框架内，学习策略网络和评估网络通过优化视觉语义激励来生成字幕，并度量图像和生成的字幕之间的相似性。关于图像字幕的生成，研究人员近期又提出了基于生成对抗网络（GAN）的文本生成模型。其中，SeqGAN（Yu 等，2017）将生成器建模为强化学习中的随机策略，如文本等离散输出，RankGAN（Lin 等，2017）提出了一种基于排序的鉴别器损失，它可以更好地评估生成文本的质量，从而产生更好的生成器。

9.4　评估指标和基准

关于自动生成字幕的质量，研究人员曾在有关自动评估和人类研究的文献中做过评估和报告。常用的自动评估标准包括双语评估替补 BLEU（Papineni 等，2002）、METEOR（Denkowski 和 Lavie，2014）、CIDEr（Vedantam 等，2015）以及 SPICE（Anderson 等，2016）。BLEU（Papineni 等，2002）在机器翻译中被广泛使用，并且度量假设和参考（或一组参考）文献之间共有的 N-Gram 分数（最多 4-Gram）。METEOR（Denkowski 和 Lavie，2014）度量的是一元模型（Unigram）的精确度和召回率，但是扩展了精确的单词匹配以包括基于 WordNet 同义词和词干标记的类似单词。CIDEr（Vedantam 等，2015）还度量了字幕假设和参考文献之间的 N-Gram 匹配，而 N-Gram 是通过 TF-IDF 进行加权的。相反，在给定参考文献的情况下，SPICE（Anderson 等，2016）可以评估图像字幕中包含语义命题内容的 F1 得分，因此与人类判断的相关性最好。这些自动评估可以有效地计算，因此大大加快了图像字幕算法的发展。然而，众所周知，所有这些自动指标只与人类的判断大致相关（Elliott 和 Keller，2014）。

截至到目前为止，研究人员已经创建了许多数据集，用于图像字幕的研究。Flickr 数据集（Young 等，2014）和 PASCAL 句子数据集（Rashtchian 等，2010）是为了促进图像字幕的研究而创建的。最近，微软公司赞助创建的 COCO（上下文中的公共对象）数据集（Lin 等，2015）是目前可供公众使用的最大图像字幕数据集。近年来，大规模数据集的可用性极大地促进了图像字幕的研究。在 2015 年，大约 15 个小组参加了 COCO 字幕挑战赛（Cui 等，2015）。挑战中的项目由人类来判断和评估。在本次比赛中，所有参赛作品的评比结果中被评比为“好”或“等于”人类水平的字幕所占比为 M1，通过图灵测试的字幕所占比为 M2。另外三个度量被用作结果的诊断和解释：1~5 级字幕的 M3-平均正确性（不正确-正确）、1~5 级字幕的 M4-平均细节量（缺乏细节-非常详细）以及与人类描述相似的 M5-百分比字幕。在评估中，每个任务都呈现出一个图像和两个字幕：一个字幕是自动生成的，另一个是人工字幕。对于 M1，需要评判者选择出哪个字幕更好地描述了图像，或者在质量相同时选择相同的选项。对于 M2，要求评判者判断出这两个字幕中哪个是由人类生成的。如果评判者选择自动生成的字幕，或者选择“无法告知”选项，就认为已经通过了图灵测试。

从 2015 年 COCO 字幕挑战中的前 15 个图像字幕系统获得了上述 M1 至 M5 指标量化的结果，以及通过自动指标度量的其他项，已在 He 和 Deng（2017）的论文中进行了总结和分析。这些系统的成功反映了我们在利用深度学习方法实现从感知到认知这一具有挑战性的任务上取得了巨大进步。

9.5 近期研究

由深度学习系统根据图像生成的自然语言字幕采用在前面提供的许多技术并提供了示例，通常它们仅给出图像内容的描述。自然语言风格在字幕生成过程中常常被忽略。具体来说，现有的图像字幕系统一直在使用语言生成模型，该模型将该风格与其他语言生成的语言风格混合，从而缺乏明确控制语言风格的机制。Gan 等人（2017a）最近的研究旨在克服这一缺陷，并进行了相关阐述。

对图像进行浪漫或幽默的自然语言描述可以极大地丰富字幕的可表达性并使其更具吸引力。一个有吸引力的图像字幕将增加图像的视觉兴趣，甚至可以成为字幕系统的一个显著特征。这对于某些应用尤其有价值，例如，增加用户在同聊天机器人“聊天”中的参与度或在社交媒体的照片字幕中启发用户。

Gan 等人（2017a）提出了 StyleNet，它能够仅使用单语言程序化语言语料库（没有配对图像）和标准事实图像/视频-字幕对来生成具有自身风格的有吸引力的图像字幕。StyleNet 是基于卷积神经网络（CNN）和循环神经网络（RNN）的结合，用于图像字幕处理。这项工作受到多任务序列对序列训练的精神激励（Luong 等，2015）。特别是，它引入了一种新的 LSTM 模型，该模型可以通过多任务训练从句子中分离出事实因素和风格因素。然后，在运行时可以显式地将风格因素合并在一起，为图像生成不同的风格的字幕。

其实，StyleNet 在新收集的 Flickr 风格化图像字幕数据集上已经进行了评估，结果表明所提出的 StyleNet 显著优于先前由自动评估和人类评估所度量的最新图像字幕方法。图 9-6 显示了一些典型的图像字幕生成实例，其中观察到具有标准事实风格的字幕仅以枯燥的语言描述图像中的事实，而浪漫和幽默风格的字幕不仅可以描述图像的内容，还以浪漫或幽默的方式表达内容。通过生成带有浪漫（例如恋爱、真爱、享受、约会等）或幽默感觉的短语可以找到一种委婉的表达方式。此外，还发现 StyleNet 生成的短语与图像的视觉内容匹配一致，使得字幕在视觉上是相关的并具有吸引力。

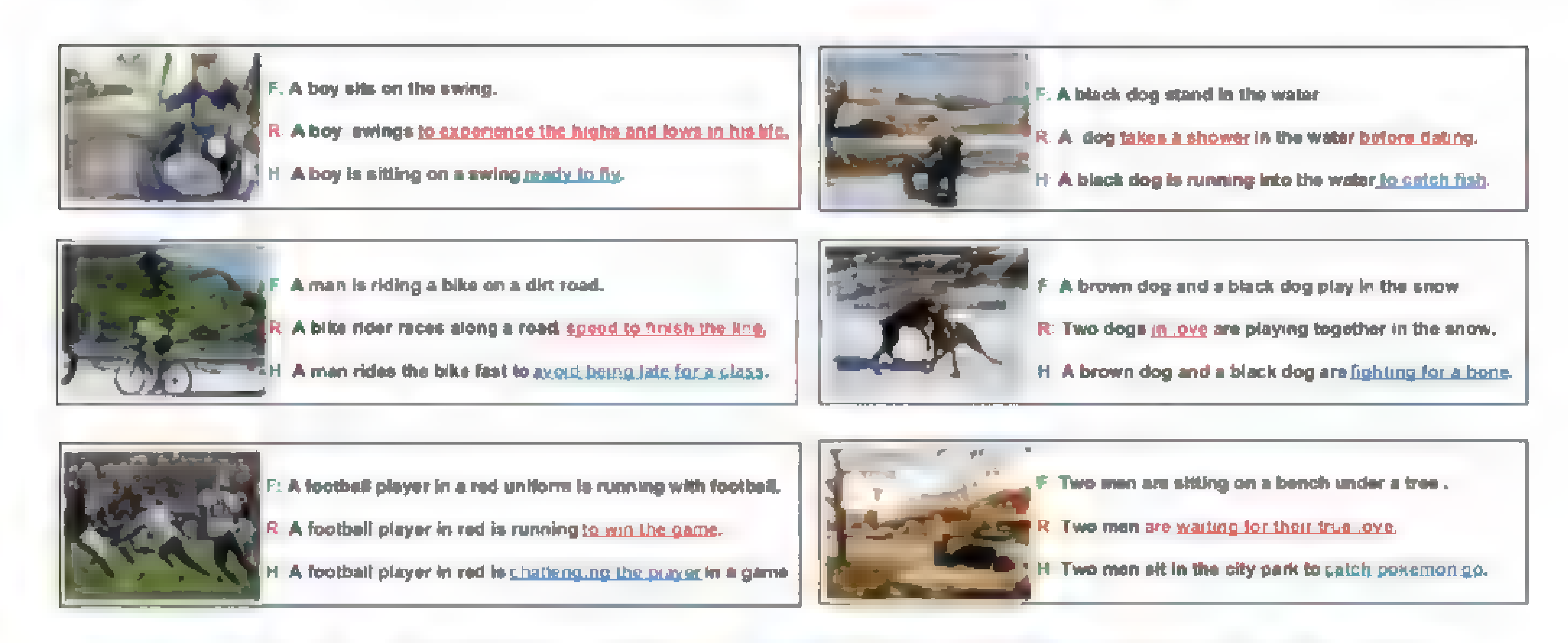


图 9-6　由 StyleNet 进行图像生成字幕的 6 个示例，每个生成的图像字幕具有 3 种不同的风格

9.6　图像字幕的产业布局

在研究界迅速发展的推动下，业界开始部署图像字幕服务。在 2016 年 3 月，微软公司向公众发布了作为云 API 的图像字幕服务。为了展示该功能的用法，它还部署了一个名为 CaptionBot（http://CaptionBot.ai）的 Web 应用程序，用于描述用户上传的任意图片。最近，微软在广泛使用的 Office 产品中部署了字幕生成服务，特别是 Word 和 PowerPoint 的应用上，用于自动生成备选文本，以便于访问。Facebook 公司也发布了一个图像字幕自动生成的工具，提供了照片中标识的对象和场景的列表。同时，Google 公司为社区开放了图像字幕自动生成系统。

通过这些在行业内大规模的部署和开源项目，实际场景中的大量图像和用户反馈正在被收集，作为不断增加的训练数据，以稳定地提高系统的性能。这将反过来激发了用于视觉理解和自然语言生成的深度学习方法的新研究。

9.7　详解图像字幕自动生成任务

9.7.1　认识数据集

首先直接和间接地了解我们正在使用的数据，是下面两个数据集：

- The ILSVRC ImageNet dataset (http://image-net.org/download)
- The MS-COCO dataset (http://cocodataset.org/#download)

我们不会直接使用第一个数据集，但它对于字幕学习至关重要。此数据集包含图像及其各自的类别标签（例如猫、狗、汽车和树等）。我们将使用一个已经在这个数据集上训练过的 CNN，所以不必从头开始下载和训练这个数据集。接下来，将使用 MS-COCO 数据集，其中包含图像及其各自的字幕。我们将使用 CNN 将图像映射到固定大小的特征向量，然后使用 LSTM 将此向量映射

到相应的字幕，直接从该数据集中学习（稍后将详细讨论该过程）。

1. ILSVRC ImageNet 数据集

ImageNet 是一个图像数据集，包含大量图像（约一百万个）及其各自的标签。这些图像属于 1 000 个不同的类别。此数据集非常具有表现力，并且在我们要为其生成字幕的图像中可以找到几乎所有的对象。因此，ImageNet 是一个很好的训练数据集，便于我们获得生成字幕所需的图像编码。我们将间接地使用这个数据集，因为我们将使用在该数据集上训练好的预训练 CNN。因此，我们不会下载这个数据集，也不会在此数据集上训练 CNN。图 9-7 显示了 ImageNet 数据集中的一些可用类别。



图 9-7 ImageNet 数据集中的一些小样例

2. MS-COCO 数据集

我们继续讨论数据集 MS-COCO（Microsoft - Common Objects in Context），这里我们将使用 2014 年的数据集。如前所述，此数据集由图像及其各自的文字描述所组成。数据集非常大（例如，训练数据集由大约 120 000 个样本组成，数据量超过 15 GB），这个数据集每年更新一次，然后举行竞赛以表彰实现最佳性能的团队。当目标是为了实现最佳性能时，使用完整数据集就显得非常有意义。然而，在我们的例子中，是需要一个模型，它可以合理地学习具有普遍性的图像字幕生成过程。因此，我们将使用较小的数据集（约 40 000 张图像和约 200 000 000 个字幕）来训练模型。图 9-8 给出了一些可用的样本。

为了学习和测试端到端图像字幕自动生成模型，我们将使用官方 MS-COCO 数据集网站上提供的 2014 年的验证数据集。该数据集由约 41 000 张图像和约 200 000 个字幕组成。我们将使用初始的 1 000 个样本作为验证集，其余的作为训练集。

提示

在实践中，我们应该使用单独的数据集进行测试和验证。但是，由于我们使用了有限的数据，为了最大化学习效果，因此这里考虑使用相同的数据集进行测试和验证。



图 9-8　MS-COCO 数据集中的一些样例

在图 9-9 中，我们可以看到验证集中的一些图像。这些图像是在验证集中精心挑选的一些的示例，展示了各种不同的对象和场景。我们将使用这些图像来检查结果，因为以可视化方式检查验证集中的 1 000 个样本是不可行的。



图 9-9　以这些图像来测试算法的图像字幕自动生成能力

9.7.2　用于图像字幕自动生成的深度学习管道

在这里，我们将看到一个高级别的图像字幕自动生成管道，然后逐个讨论它，直到得到一个完整的模型。图像字幕自动生成框架由三个主要组件和一个可选组件组成：

- CNN 生成用于图像的编码向量。
- 词嵌入层学习词向量。
- （可选）可以将给定词向量维度转换为任意维度的适配层（后续将进行详细讨论）。

- LSTM 获取图像的编码向量，并输出相应的字幕。

首先，让我们看一下 CNN 生成图像的编码向量。我们可以通过在大型分类数据集（例如 ImageNet）上训练 CNN 并使用该知识生成图像的压缩向量化表示来实现这一点。

有人可能会问，为什么不将图像输入 LSTM 呢？让我们回到第 8 章中提到的一个简单计算：“如果输入层增加了 500 个单位，就会导致参数增加 200 000 个”。

我们在这里处理的图像大约在 150 000~224×224×3 之间，这应该可以让你意识到会导致 LSTM 参数数量增加的程度。因此，对于压缩向量化表示的工作变得至关重要。LSTM 不适合直接处理原始图像数据的另一个原因是，与使用 CNN 处理图像数据相比，它显得过于复杂。

提示

其实，LSTM 中也存在着卷积 LSTM（Convolution LSTM）这样的变体。卷积 LSTM 可以使用卷积运算代替全连接图层去处理图像输入。这种网络大量用于时空问题（例如，天气数据或视频预测），其具有数据的空间和时间维度。我们可以从 Jeff Donahue 等人的论文《*Long-term Recurrent Convolutional Networks for Visual Recognition and Description*》（Proceedings of the IEEE conference on Computer Vision and Pattern Recognition，2015）中查阅有关卷积 LSTM 的更多信息。

虽然训练过程完全不同，但是这个训练过程的目标与学习词向量的目标是相似的。对于词向量，我们希望相似的词具有相似的向量（高相似度），而不同的词具有不同的向量（低相似度）。换句话说，如果$Image_x$表示针对图像x获得的编码向量，那么以下式子应该成立：

$$Distance(Image_{cat}, Image_{volcano}) > Distance(Image_{cat}, Image_{dog})$$

接下来，我们将通过从 MS-COCO 数据集提供的所有字幕中提取所有单词来学习被创建的文本语料库的词向量。学习词向量有助于我们减少 LSTM 输入的维度，还有助于生成更有意义的特征作为 LSTM 的输入。这是机器学习管道中的另一个关键目标。

当我们使用 LSTM 生成文本时，使用单词的 One-Hot 编码表示或词向量表示。因此，LSTM 的输入始终是固定大小的。如果输入大小是动态变化的，我们很难使用标准 LSTM 来处理它。其实没有必要担心这一点，因为我们只处理文本信息。

然而，在这种情况下，我们在处理图像和文本时需要确保图像的编码向量和对应于图像字幕的每个词的表示都具有相同的维度。此外，使用词向量可以为所有单词创建任意固定长度的表示方式。因此，我们使用词向量来匹配图像编码向量长度。

最后，我们将为每个图像创建一个数据序列，其中序列的第一个元素是图像的向量化表示，然后是图像字幕的词向量。我们将使用这个数据序列来训练 LSTM。

这种方法类似于论文《*Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*》（Xu 等，第 32 届机器学习国际会议论文集，2015）中发现的方法，相关处理过程如图 9-10 所示。

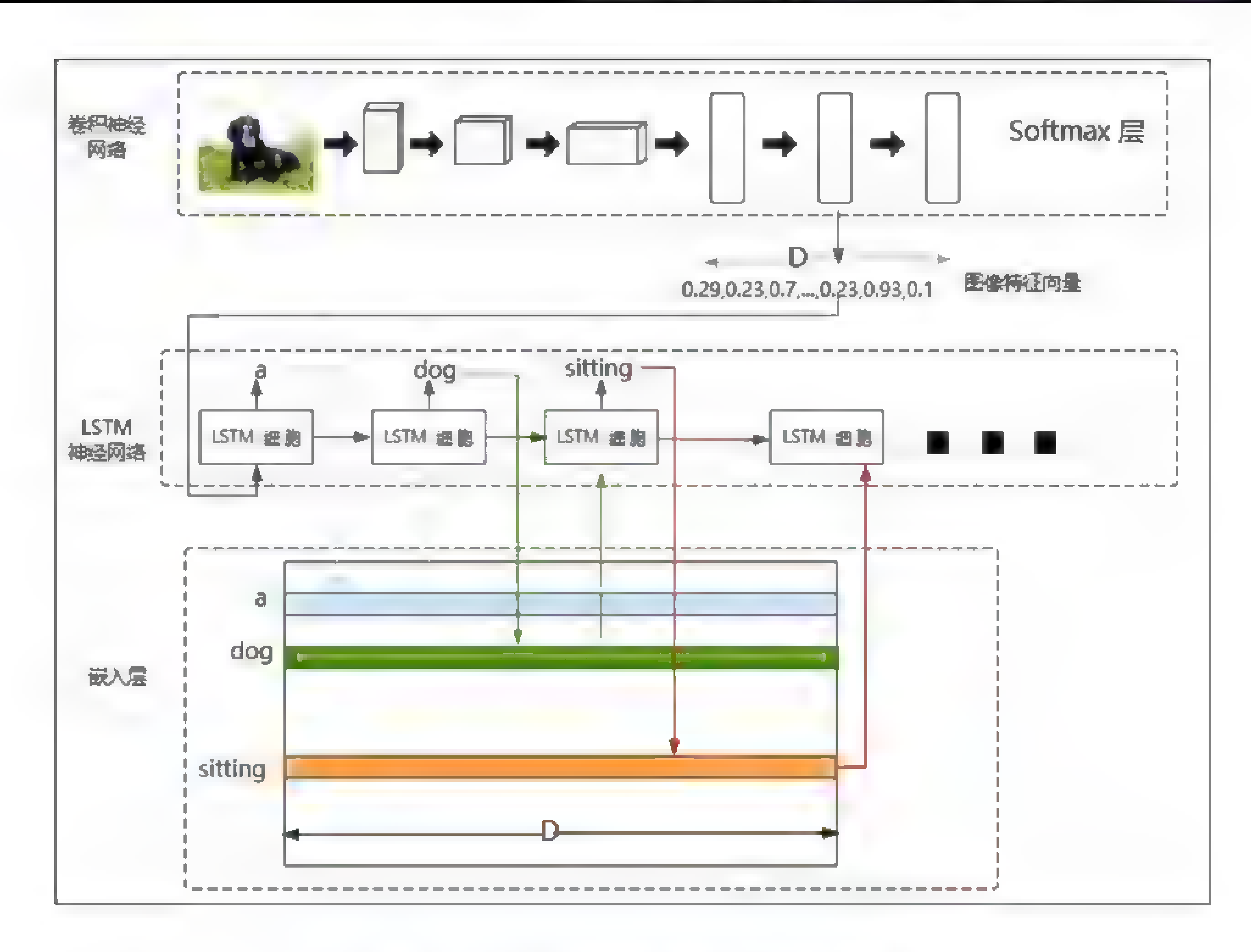


图 9-10　用于生成图像字幕的机器学习管道

9.7.3　使用 CNN 提取图像特征

下面详细讨论如何使用 CNN 来提取图像的特征向量。为了获得良好的特征向量，我们首先需要利用图像及其对应的图像类别训练 CNN 或使用互联网上免费提供的预训练好的 CNN。如果我们从头开始训练 CNN，将不得不进行重复性的工作，我们这里选择预训练好的模型（可供免费下载）。另外需要说明的是，如果 CNN 需要能够描述多个对象，那么需要在对应于各种对象的一组分类上进行训练。这就是在大型数据集（如 ImageNet）上训练模型很重要的原因所在（例如，与只有 10 个不同类别的小型数据集的训练相比）。如前所述，ImageNet 包含 1 000 个对象类别。这对我们试图解决的任务来说已经足够了。

不过，ImageNet 包含约 1MB 的图像。此外，由于有 1000 个类，显然不能使用具有简单结构的小型 CNN（例如，具有几层的 CNN）来进行学习。我们需要更强大、更深层次的 CNN，但是由于 CNN 的复杂性和数据集本身的复杂性，在 GPU 上训练这样的神经网络可能需要几天（甚至几周）的时间。例如，VGG（即一个著名的 CNN，在 ImageNet 上给出了非常好的图像分类准确性）可能需要 2~3 周的时间来进行训练。

因此，我们需要更巧妙的方法来解决这个问题。幸运的是，像 VGG 这样的 CNN 可以随时下载，所以我们可以在没有进行任何单独训练的情况下使用它们，这就是它被称为预训练模型的原因所在。使用预训练模型可以使我们节省数周的时间，因为我们所需要的只是学习的权重值和 CNN

的实际结构，以重建网络并立即使用它进行推理工作。

本实例中，我们将使用 VGG CNN（可在 http://www.cs.toronto.edu/~frossard/post/vgg16/处获得）。VGG 架构在 2014 年的 ImageNet 竞赛中获得第二名，且 VGG 有几种变体：13 层深度网络（VGG-13）、16 层深度网络（VGG-16）和 19 层深度网络（VGG-19）。我们这里将使用 VGG-16，其网络架构如图 9-11 所示。

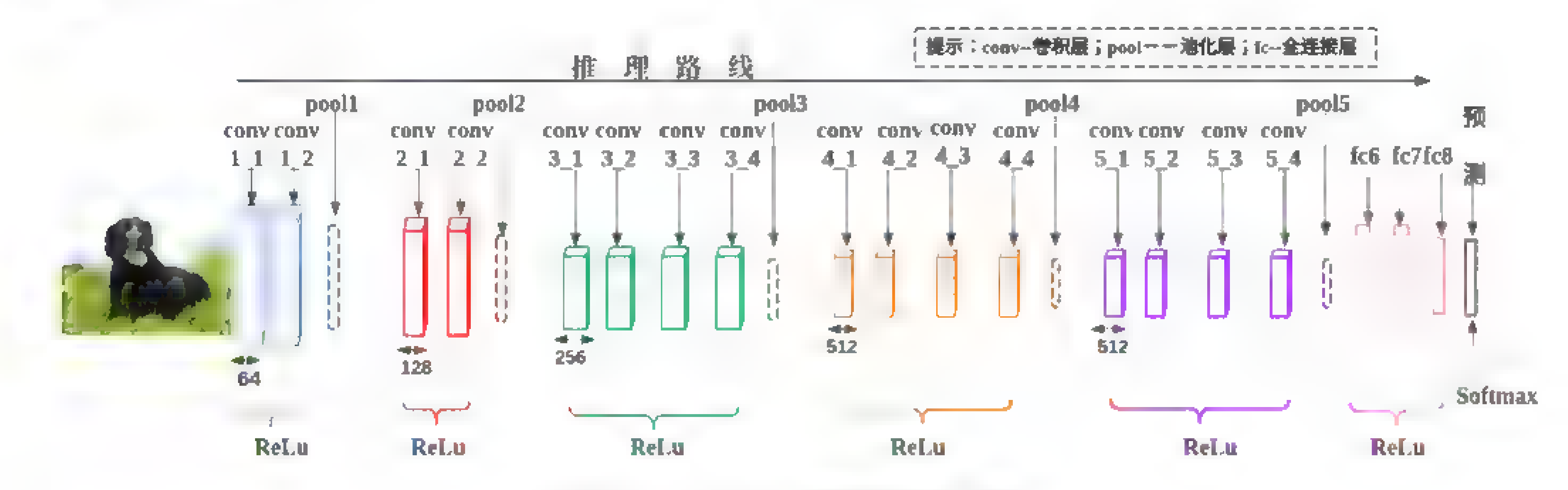


图 9-11 VGG-16 网络架构示意图

9.7.4 使用 VGG-16 加载权重值并进行推理

网站 http://www.cs.toronto.edu/~frossard/post/vgg16/提供了权重值来作为 NumPy 数组的字典，且有 16 个权重值和 16 个偏差值对应于 16 层的 VGG-16 模型，它们分别保存在下面一些字段中：

```
conv1_1_W, conv1_1_b, conv1_2_W, conv1_2_b, conv2_1_W, conv2_1_b…
```

首先，从网站上下载文件并将其保存至 ch9/image_caption_data 文件夹中。现在我们将讨论具体实现过程：从加载下载得到的 CNN 到使用预训练的 CNN 进行预测。首先，创建必要的 TensorFlow 变量并用下载的权重值来加载它们。其次，定义一个输入的读取管道，以读取图像来作为 CNN 的输入以及几个预处理步骤。然后定义 CNN 的推理操作以获得相关输入的预测。接着定义具体计算以获得图像分类，并为 CNN 定义认为符合给定输入的图像的最佳分类预测。虽然最后一个操作不需要为图像生成字幕，但是正确地配置预训练的 CNN 非常重要。

1. 构建和更新变量

首先使用以下内容将包含 CNN 权重值的 NumPy 数组字典加载到内存中（即加载到 CNN 的记忆中）：

```
weight_file = os.path.join('image_caption_data', 'vgg16_weights.npz')
weights = np.load(weight_file)
```

接下来，创建 TensorFlow 变量并为它们赋予实际的权重值。此外，这可能占用相当多的内存。因此，为了避免机器崩溃，我们将特别要求 TensorFlow 将其保存在 CPU 而不是 GPU 上。这里将简述用于构建和加载具有正确权重值的 TensorFlow 变量的代码。我们将首先在 Python 列表 TF_SCOPES 中定义所有字典的键（表示 CNN 的不同层 ID）。然后将迭代遍历每个图层 ID，同时

使用相应的权重值矩阵和偏差向量作为初始化过程，根据相应图层 ID 命名具体的 TensorFlow 变量：

```
def build_vgg_variables(npz_weights):
        '''
构建所需的 TensorFlow 变量以填充 VGG-16 并用实际权重值填充它们
        :param npz_weights: loaded weights as a dictionary
        :return:
        '''

        params = []
        print("Building VGG Variables (TensorFlow)...")
        with tf.variable_scope('CNN'):
            # 迭代每个卷积和全连接层，并使用变量作用域来创建 TensorFlow 变量
            for si,scope in enumerate(TF_SCOPES):
              with tf.variable_scope(scope) as sc:
               weight_key, bias_key = TF_SCOPES[si]+'_W', TF_SCOPES[si]+'_b'

              with tf.device('/cpu:0'):
               weights = tf.get_variable(TF_WEIGHTS_STR,
                initializer= npz_weights[weight_key])
bias = tf.get_variable(TF_BIAS_STR, initializer = npz_weights[bias_key])
    params.extend([weights,bias])

return params
```

2. 预处理输入

接下来，定义一个输入管道以便将图像输入到 VGG-16 模型中。VGG-16 对输入图像有以下要求，以使预测趋于正确：

- 输入量应为[224,224,3]。
- 输入应该是零均值（但不是单位方差）。

以下代码创建一个管道，该管道直接从一组给定的文件名中读取，且应用前面的转换并创建一批此类转换后的图像。此过程在代码文件里由 preprocess_inputs_with_tfqueue 函数进行定义。

首先，我们将定义一个文件名队列，这里保存了我们应该读取的文件名（图像的文件名）：

```
# 文件名的 FIFO 队列
# 创建 FIFO 队列，以便 reader 可以读取队列
filename_queue = tf.train.string_input_producer(filenames, capacity=10,
shuffle=False)
```

其次，这里将定义一个读取器，它将文件名队列作为输入并输出到一个缓冲区中，而后从该缓冲区读取文件名以找到和读取对应的图像数据：

```
#读取器（reader）采用文件名队列和 read()逐个输出数据
reader = tf.WholeFileReader()
        , image buffer = reader.read(filename queue,name='image read op')
# 读取原始图像数据并作为 unit8 返回
dec image = tf.image.decode jpeg(contents= image buffer,channels=3,
name='decode jpg')
# 将 unit8 数据转换为 float32 类型
float image = tf.image.convert image dtype(dec image,
                              dtype=tf.float32,name= 'float_image')
```

接下来进行预处理工作：

```
# 将图像大小调整为 224×224×3
resized_image = tf.image.resize_images(float_image,[224,224])*255.0
# 对于 VGG 模型，图像仅是零均值的（不是标准化的单位方差）
std_image = resized_image - tf.reduce_mean(resized_image,axis=[0,1],
keepdims=True)
```

在定义预处理管道之后，我们将要求 TensorFlow 一次生成一批预处理的图像：

```
image_batch = tf.train.batch([std_image], batch_size = batch_size,
capacity = 10, allow_smaller_final_batch=False,name='image_batch')
```

3. 推理

至此，我们已经创建了 CNN，且已经定义了一个管道用于读取图像，通过读取保存在磁盘上的图像文件来创建批处理。现在我们想用从管道读取的图像来推断 CNN。推理是指传递输入（图像）并获得预测（属于某类图像的概率）作为输出。为此，我们将从第一层开始并迭代到 softmax 层。此过程在代码文件里由 inference_cnn 函数来定义。

在每一层，我们将获得权重值和偏差，代码如下：

```
def inference_cnn(tf_inputs, device):
with tf.variable_scope('CNN'):
      for si, scope in enumerate(TF_SCOPES):
           with tf.variable_scope(scope,reuse=True) as sc:
                             weight, bias = tf.get_variable(TF_WEIGHTS_STR),
                                            tf.get_variable(TF_BIAS_STR)
```

然后对于第一个卷积层进行计算并输出：

```
h = tf.nn.relu(tf.nn.conv2d(tf_inputs,weight,strides=[1,1,1,1],
padding='SAME')+bias)
```

对于其余的卷积层进行计算并输出，其中的输入是前一层的输出：

```
h = tf.nn.relu(tf.nn.conv2d(h, weight, strides [1, 1, 1, 1],padding='SAME')
+ bias)
```

对于池化层，计算如下：

```
h = tf.nn.max_pool(h,[1,2,2,1],[1,2,2,1],padding='SAME')
```

然后，对于紧跟在最后一个卷积层/池化层之后找到的第一个完全连接的层，我们将给出如下定义的输出。我们需要将来自[batch size，height，width，channels]的最后一个卷积层/池化层的输入重塑为[batch size，height * width * channels]大小，因为接下来就是一个完全连接层：

```
h_shape = h.get_shape().as_list()
h = tf.reshape(h,[h_shape[0], h_shape[1] * h_shape[2] * h_shape[3]])
h = tf.nn.relu(tf.matmul(h, weight) + bias)
```

对于除最后一层之外的下一组完全连接层，我们得到如下输出：

```
h = tf.nn.relu(tf.matmul(h, weight) + bias)
```

最后，对于最后一个完全连接层，我们不进行任何类型的激活操作。这将是我们提供给 LSTM 的图像特征表示，它是一个 1000 维的向量：

```
out = tf.matmul(h,weight) + bias
```

4.提取图像的向量化表示

我们从 CNN 中提取的最重要的信息是图像的特征表示。对于图像的特征表示，我们将在应用 softmax 预测之前获得最后一层的网络输出。因此，对应于单个图像的向量长度为 1 000：

```
tf_train_logit_prediction = inference_cnn(train_image_batch, device)
tf_test_logit_prediction = inference_cnn(test_image_batch, device)
```

5. 使用 VGG-16 预测图像分类的概率

接下来，我们将定义获取图像特征表示所需的操作以及实际的 softmax 预测，以确保我们的模型在实际应用中的正确性。我们将为训练数据和测试数据定义这些相关操作：

```
tf_train_softmax_prediction = tf.nn.softmax(tf_train_logit_prediction)
tf_test_softmax_prediction = tf.nn.softmax(tf_test_logit_prediction)
```

现在运行这些操作，看看它们是否正常工作，如图 9-12 所示。

现在来看，我们的 CNN 似乎知道它在做什么。当然，也存在错误分类的样本（例如，长颈鹿被识别为美洲驼），但大多数情况下是正确的。

提示

在运行前面已定义的操作以获取特征向量和预测时，请注意 batch size 变量。增加此值将使代码快速运行。但是，如果没有足够大的内存（大于 8 GB），就可能导致系统崩溃。如果你没有高端的计算机，建议将此值保持在 10 以下。



图 9-12 利用 VGG 对测试图像进行分类预测

9.7.5 学习词向量

本小节讨论如何为图像字幕数据集中的单词执行学习词向量的操作。首先，我们对字幕进行预处理：

```
def preprocess_caption(capt):
      capt = capt.replace('-',' ')
      capt = capt.replace(',','')
      capt = capt.replace('.','')
      capt = capt.replace('"','')
      capt = capt.replace('!','')

      capt = capt.replace(':','')
      capt = capt.replace('/','')
      capt = capt.replace('?','')
      capt = capt.replace(';','')
      capt = capt.replace('\' ',' ')
      capt = capt.replace('\n',' ')
      return capt.lower()
```

例如，请考虑以下句子：

```
A living room and dining room have two tables, couches, and multiple chairs.
```

这将转换为以下内容：

```
a living room and dining room have two tables couches and multiple chairs
```

然后使用连续词袋（CBOW）模型来学习词向量，就像我们在第 4 章词嵌入学习词向量中所做的那样。在学习词向量时，我们需要谨记一点：词向量的维度应该与为图像获得的特征表示的维度

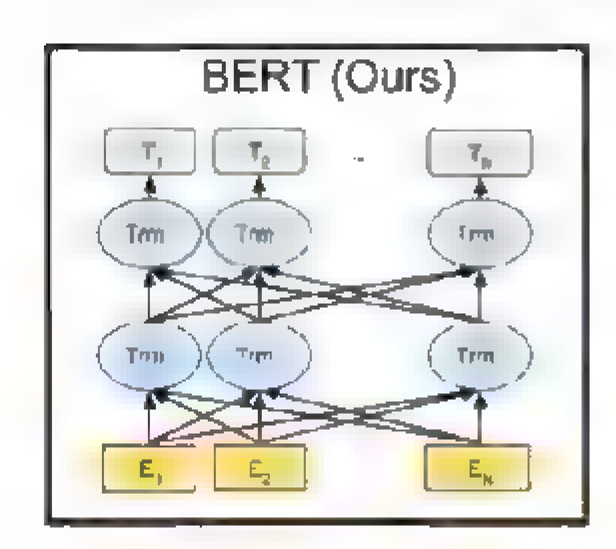
11.12.2　BERT 模型结构

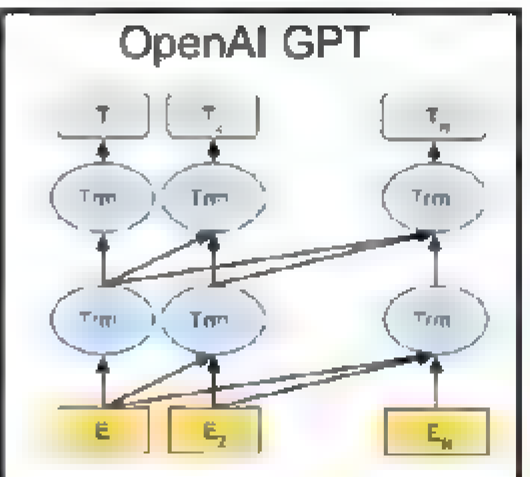
这里根据 Jacob Devlin 等人的论文《*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*》进行模型简要介绍。其实，BERT 模型沿袭了 GPT 模型的结构，采用 Transformer 的编码器作为主体模型结构，也就是论文中提到的基于 Ashish Vaswani 等人（2017）的《*Attention is all you need*》中描述的原始实现的 Multi-layer Bidirectional Transformer 编码器，并在 tensor2tensor 库中进行发布。而 Transformer 则摒弃了 RNN 的循环式网络结构，仅是基于注意力机制来对一段文本进行建模。由于现在 Transformer 的使用变得无处不在，论文中的实现与原始实现完全相同，因此这里将省略对模型结构的详细描述。

在这项工作中，论文将层数（Transformer Blocks）表示为 L，将隐藏大小表示为 H，将 Self-Attention Heads 的数量表示为 A。在所有情况下，将 Feed-Forward/Filter 的大小设置为 4H，即 H = 768 时为 3072，H = 1024 时为 4096。论文主要给出了以下两种模型大小的结果。

- BERT BASE：L=12, H=768, A=12, Total Parameters=110M
- BERT LARGE：L=24, H=1024, A=16, Total Parameters=340M

为了进行比较，论文选择了与 OpenAI GPT 具有相同的模型大小。然而，重要的是 BERT Transformer 使用了双向 Self-Attention（自注意力）机制，而 GPT Transformer 则使用受限制的 Self-Attention 机制，其中每个 Token（切分词）只能处理其左侧的上下文。双向 Transformer 通常被称为"Transformer Encoder"，而左侧上下文则被称为"Transformer Decoder"，因为它可以用于文本生成。BERT、OpenAI GPT 和 ELMo 之间的比较如图 11-28 所示。





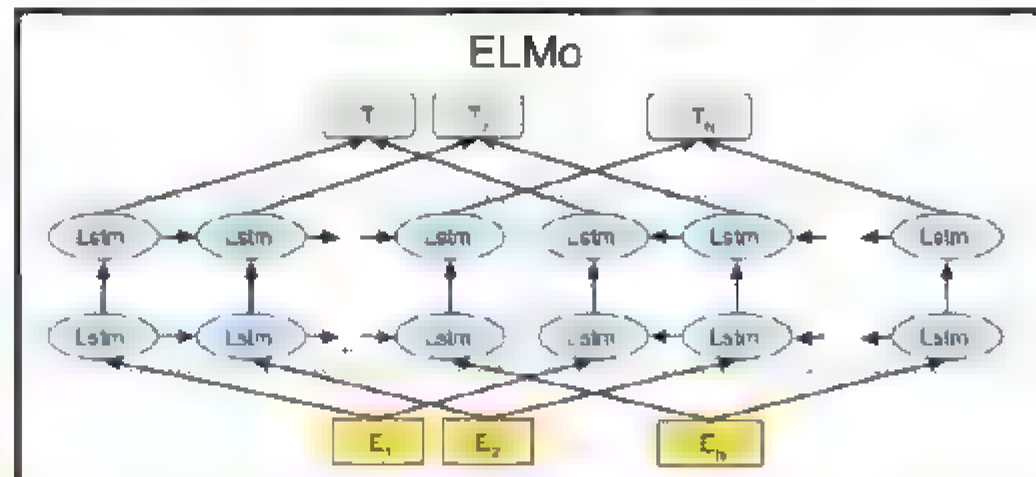


图 11-28　预训练模型架构的差异

BERT 使用双向 Transformer；OpenAI GPT 使用从左到右的 Transformer；ELMo 使用经过独立训练的从左到右和从右到左 LSTM 的串联来生成下游任务的特征。三个模型中，只有 BERT 表示在所有层中共同依赖于左右上下文。

BERT 对 GPT 的第一个改进就是引入了双向的语言模型任务。此前其实也有一些研究在语言模型这个任务上使用了双向的方法，例如在 ELMo 中是通过双向的两层 RNN 结构对两个方向进行建模，但两个方向的损失计算是相互独立的。而 BERT 的作者指出：这种两个方向相互独立或只有单层的双向编码可能没有发挥出最好的效果，我们不仅需要双向编码，还需要加深网络的层数。但加深双向编码网络却会引入一个问题，导致模型最终可以间接地"窥探"到需要预测的词。这个"窥探"的过程可以用图 11-29 来表示。

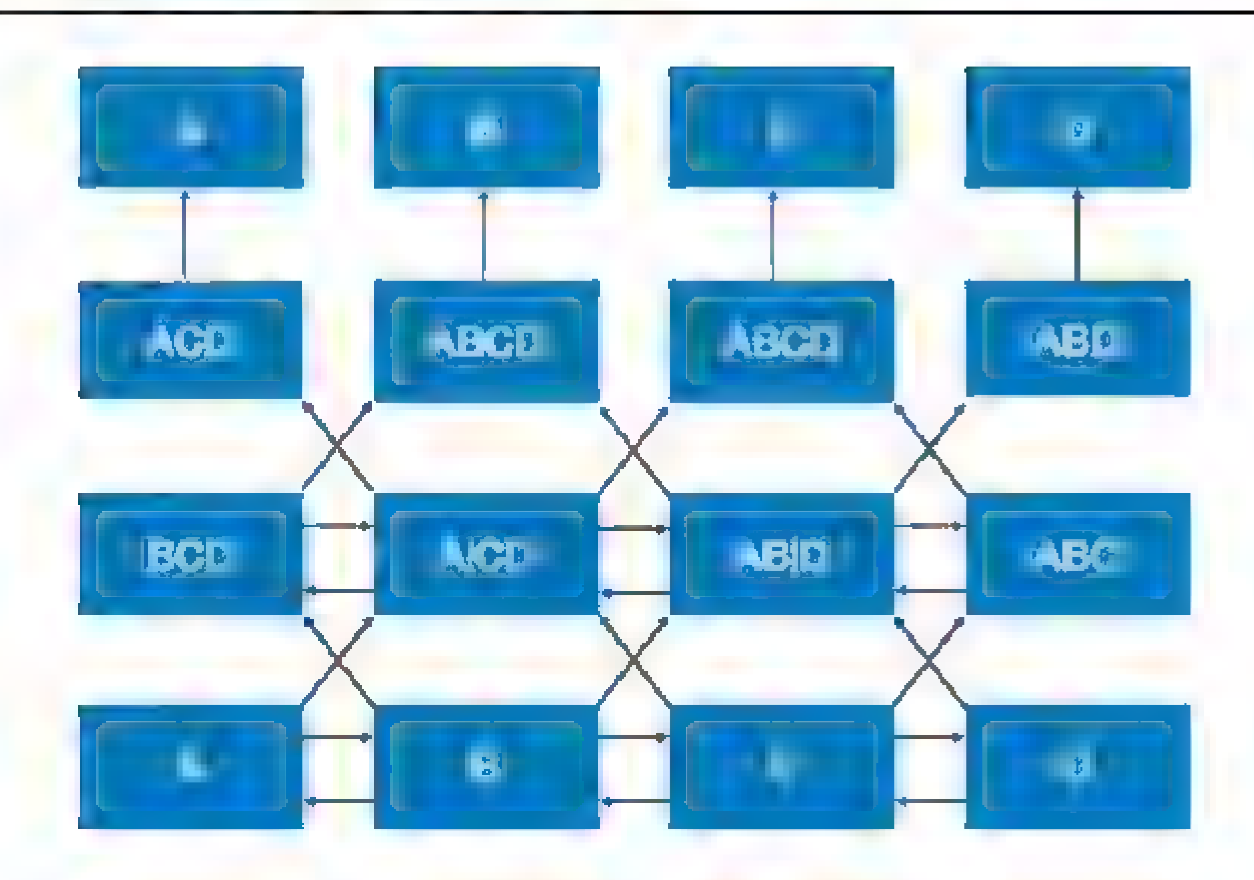


图 11-29 模型间接“窥探”到需要预测词的过程之示意图

从图 11-29 中可以看到经过两层的双向操作，每个位置上的输出已经带有原本这个位置上词的信息了。这样的“窥探”会导致模型预测词的任务变得失去意义，因为模型已经看到每个位置上是什么词了。

为了解决这个问题，我们可以从预训练的目标入手。我们想要的其实是让模型学会某个词适合出现在怎样的上下文语境中；反过来说，如果给定了某个上下文语境，那么我们希望模型能够知道这个地方适合填入怎样的词。从这一点出发，其实可以直接去掉这个词，只让模型看上下文，然后来预测这个词，但这样做会丢掉这个词在文本中的位置信息。还有一种方式是在这个词的位置上随机输入某一个词，但如果每次都随机输入，就可能会让模型难以收敛。

BERT 的作者提出了采用 MaskLM 的方式来训练语言模型。通俗地说，就是在输入一句话的时候，随机地选一些要预测的词，然后用一个特殊的符号来代替它们。尽管模型最终还是会看到所有位置上的输入信息，但由于需要预测的词已经被特殊符号代替，所以模型无法事先知道这些位置上是什么词，这样就可以让模型根据所给的标签去学习这些地方该填的词了。

然而这里还有一个问题，就是我们在预训练过程中所使用的这个特殊符号，在后续的任务中是不会出现的。因此，为了与后续任务保持一致，作者按一定的比例在需要预测的词位置上输入原词或某个随机的词。当然，由于一次输入的文本序列中只有部分的词用来进行训练，因此 BERT 在效率上会低于普通的语言模型，作者也指出 BERT 的收敛需要更多的训练步数。

BERT 另外一个创新是，在双向语言模型的基础上额外增加了一个句子级别的连续性预测任务。这个任务的目标也很简单，就是预测输入 BERT 的两端文本是否为连续的文本，作者指出引入这个任务可以更好地让模型学到连续的文本片段之间的关系。在训练的时候，输入模型的第二个片段会以 50%的概率从全部文本中随机选取，剩下 50%的概率选取第一个片段的后续文本。

11.13 总结

在本章中，我们首先详细介绍了基于规则的机器翻译、统计机器翻译等传统的机器翻译的情况，包括基于规则的机器翻译和统计机器翻译各自的研究背景、种类和模型原理。

其次，我们详细讨论了神经网络机器翻译（NMT）。在该部分对神经网络机器翻译模型的架构和工作机制做了详细解读，并通过一个小实例进行了深入阐述。我们讨论了这些系统的基本概念，并将模型分解为词嵌入层、编码器、上下文向量和解码器。我们之所以首先介绍了词嵌入层的好处，是因为与独热（One-Hot）编码向量相比，它提供了词的语义表示。其次我们了解了编码器的目标，即学习一个代表源语句的固定长度的向量。接下来，学习了固定长度的上下文向量，再用它来初始化解码器。解码器负责产生源语句的实际翻译。然后我们讨论了神经网络机器翻译系统中的训练和推理是如何工作的。

接着，我们给出了评估机器翻译系统的重要指标：BLEU 评分。这里涉及 BLEU 的度量、BLEU 的调整和惩罚因子、BLEU 得分总结等。

然后，我们研究了神经网络机器翻译系统的实现，该系统将句子从德语翻译成英语，以此理解神经网络机器翻译系统的内部机制。在这里，我们研究了使用基本 TensorFlow 操作实现的神经网络机器翻译系统，因为与使用 TensorFlow 中的现成库（如 seq2seq）相比，这使我们能够深入了解系统的逐步执行。

最后，我们讨论了引爆 2018 年的重大研究发现：BERT 模型。我们了解到 BERT 模型是第一个用于预训练自然语言处理（NLP）任务的无监督、深度双向的系统（Unsupervised, Deeply Bidirectional System），且该模型在 11 项 NLP 任务中夺得 STOA 结果，令人惊喜。在这里，我们知道 BERT 模型是经过改进后的 GPT 模型，其中的两个创新之处是：采用 MaskLM 的方式来训练语言模型；在双向语言模型的基础上额外增加了一个句子级别的连续性预测任务。

在下一章中，我们将讨论智能问答系统。

第 12 章

智能问答系统

问答（Question Answering，QA）是自然语言处理中一项具有挑战性的任务。近年来，随着深度学习在语义和句法分析、机器翻译、关系抽取等自然语言处理任务上取得了显著的成功，利用深度学习来完成问答任务也得到了越来越多的关注。本章将简要介绍深度学习方法在两个典型问答任务上的最新进展的情况。（1）基于知识库深度学习的问答（KBQA），主要采用深度神经网络来理解问题的含义，并尝试将其转化为结构化查询，或者直接转化为分布式语义表示。（2）机器理解中的深度学习（Machine Comprehension，MC）是一种基于新型神经网络的端到端范式，用于直接计算问题、答案和给定段落之间的深层语义匹配。

12.1 概要

从简单的文档检索到自然语言问答（QA），网络搜索正处于深刻变革的前沿中。它需要准确理解用户在自然语言问题方面的含义，从网络上各种信息中提取有用的事实，并选择出合理的答案。与其他自然语言处理（NLP）任务，如词性标注、解析和机器翻译类似，大多数传统的 QA 方法都是基于符号表示的。在这样的范式中，问题和答案中的所有元素，包括单词、短语、句子、文档等，通常都是借助于 NLP 基本模块来处理的，然后转换为某些结构化或非结构化格式，如词袋、解析树、逻辑形式等。在给定的文档或网页中，计算问题和候选答案之间语义的相似性或相关性，并且具有最高分数的候选项便是最终答案。尽管如此，这种范式的弱点在于所谓的“语义鸿沟”（Semantic Gap），即具有相似含义的文本跨度可能会用不同的符号表示。

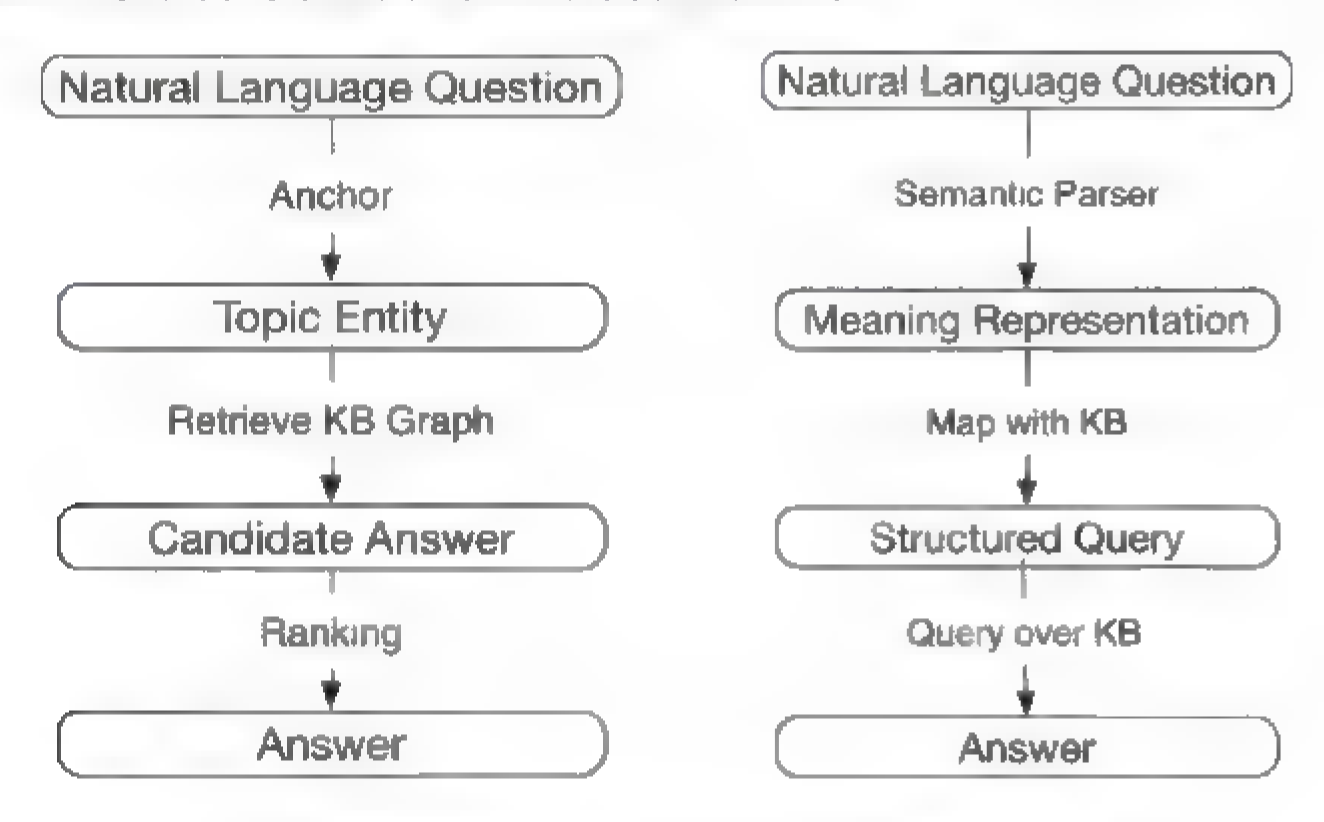
在神经网络模型中，文本通常表示为分布式向量，而文本跨度之间的精确匹配可以由分布式向量之间的运算来实现。这样传统方法中的语义鸿沟问题就可以在一定程度上得到缓解。

此外，问答任务中还有几个分支，包括基于检索的问答（IRQA）、社区的问答（CQA）、基于知识库的问答（KBQA）和机器理解（MC）的问答。在这里，我们主要关注基于知识库的问答（KBQA）和基于机器理解（MC）的问答，因为这两个问答需要对文本从问题到文档进行更多的

语义分析和理解。另外，我们还将从两个角度讨论 KBQA 的最新进展，并进一步回顾针对机器理解（MC）的深度学习工作等。

12.2 基于知识库的问答

截至目前，其实已经有研究人员在扩展新的神经网络模型方面进行了许多成功的尝试，并且提高了知识库问答系统的性能。这里有多种新型的神经网络组件或架构及其变体，如 CNN、RNN（LSTM、BLSTM）、注意力机制和记忆网络，并且已经在任务中得到了检验。这些工作内容可以被分为两种模式：信息抽取（Information Extraction，IE）和语义分析（Semantic Parsing，SP），如图 12-1 所示。信息抽取模式通常使用各种关系提取技术从知识库中检索一组候选答案，然后将这些候选答案与压缩特征空间中的问题进行比较；语义分析模式则是借助新型的组件或神经网络结构，从句子中提取出正式的/符号化的表示或结构化的查询。



（a）信息抽取模式框架（b）语义分析模式框架

图 12-1

从另一个角度来看，我们可以将有关利用深度学习方法来促进基于知识库的问答（KBQA）开发工作分为两类：在传统的 KBQA 框架内使用新型的神经网络模型来改进特定的组件和在统一的神经网络架构中标准化工作任务。前一种观点主要侧重于利用先进的神经网络模型来改进现有的组件，如特征提取、关系识别、语义匹配或相似度计算等；后者则是强调利用新型的深度学习框架来使自然语言的问题和候选答案被构建在同一低维语义空间内。因此，该 KBQA 任务可以转换为问题的向量和该空间中候选答案之间相似度的计算问题，通常以信息抽取的方式展开。

12.2.1 信息抽取

我们在使用深度学习方法的过程中，主要是侧重于寻找更好的方法将自然语言的问题和来自知识库的候选答案嵌入同一个且被压缩的语义空间中。一般情况下，这些工作会在统一的神经网络架构中以“检索-嵌入-比较”（Retrieval Embedding–Comparing）流水线式（有时也称为管道）的形

式对解决方案进行标准化。

1. 基本介绍

信息抽取（Information Extraction，IE）是把文本中包含的信息进行结构化处理，变成表格一样的组织形式。输入信息抽取系统的是原始文本，输出的是固定格式的信息点。信息点首先从各种各样的文档中被抽取出来，然后以统一的形式集成在一起，这就是信息抽取的主要任务。信息以统一的形式集成在一起的好处是方便检索和比较。信息抽取技术并不试图全面理解整篇文档，只是对文档中包含相关信息的部分进行分析，至于哪些信息是相关的，将由系统设计时指定的范围来决定。

信息抽取技术对于从大量的文档中抽取需要的特定事实来说是非常有用的，互联网就是这么一个文档库。在互联网上，同一主题的信息通常分散存放在不同的网站上，表现形式也各不相同，如果能将这些信息收集在一起，并利用结构化形式存储，那将是非常有益的。

由于互联网上的信息载体主要是文本，所以信息抽取技术对于那些把互联网当成知识来源的人来说是至关重要的。信息抽取系统可以看作是把信息从不同文档中转换成数据库记录的系统，成功的信息抽取系统将把互联网变成巨大的数据库。

虽然信息抽取技术是近年来发展起来的新技术领域，但也面临着许多新的挑战。信息抽取原来的目标是从自然语言文档中找到特定的信息，是自然语言处理领域特别有用的一个子领域。它所开发的信息抽取系统既能处理含有表格信息的结构化文本，又能处理自由式文本（如新闻报道）。信息抽取系统中的关键组成部分是一系列的抽取规则或模式，其作用是确定需要抽取的信息。互联网上文本信息的大量增加导致这方面的研究得到高度重视。

2. 向量表示

Bordes 等人于 2014 年率先提出了信息抽取模式的方法，他们没有单独将类别、实体引用和关系模式映射到知识库中对应的类型、实体和谓词，而是提出了一种更直接的方法。他们设计了一个联合嵌入框架来学习结构化知识库中的单词、实体、关系和其他语义项的向量表示，并设法将自然语言问题映射到知识库中的子图，如图 12-2 所示。

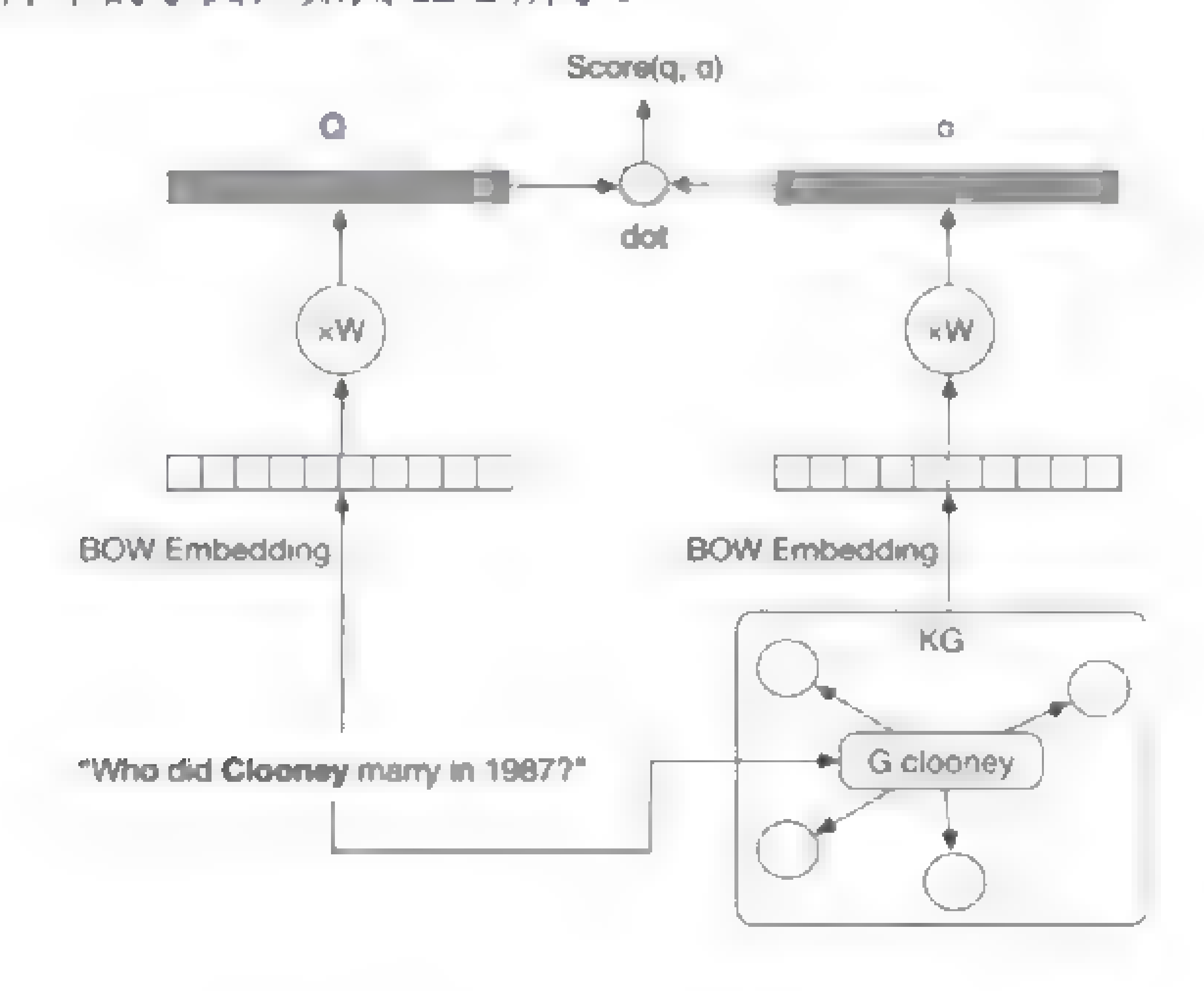


图 12-2 向量表示的示意图

当自然语言问题和候选子图用低维向量表示时，可以很容易地计算出问题和子图之间的相似性。该模型需要带注释的问答配对作为训练数据，也可以通过简单的模式和多任务范式自动收集更多的训练实例。通过同时优化其他资源或相关的辅助任务，如转述任务，该模型旨在确保相似的话语具有更高的相似性，从而减轻客服中心对人力的需求。

图 12-2 所示的框架同样也遵循简单明了的流水线式（有时也叫管道）结构，即检索-嵌入-比较，而不依赖于人工构造的特征、额外的句法分析或传统抽取模型所采用的经验规则那样，并在基准数据集上实现了具有竞争力的性能。

为了便于实现，首先用词袋模式来表示，然后经过一个压缩过程，这个过程忽略了自然语言问题中的句法结构。类似的方法也适用于候选答案，其中子图简单地表示为其所涉及实体和关系的 Multi-Hot 形式。这种简化的方法其实是防止模型在自然语言表示或知识库本身使用更多的信息来源，例如问题中的关系短语或答案类型指示器，或者知识库中的实体谓词一致性。

此外，目前神经网络模型的处理方法还不能很好地处理语义的复合性，以及除了词袋或实体袋关系表示之外的各种约束，如“小明的父亲的母亲的儿子”和“小明的母亲的父亲的儿子”。而在某种程度上，可以通过对问题进行更深入的语法分析，或者频繁地对知识库进行结构挖掘来解决。

2. 使用 CNN 嵌入特征

Yih 等人（2014）提出的是利用卷积神经网络（CNN）来讲解决单一关系问题，这与 Bordes 等人在词袋模型里处理所有问题的模式是不同的。在 Yih 等人的文章中，实际上是使用基于 CNN 的语义模型（CNNSM）构造两个不同的映射模型：一个用于标识问题中的实体；另一个用于将关系映射到知识库的关系。这里需要说明的是，假设目标问题只包含一个实体和一个关系，这确实占据了各种 KBQA（基于知识库的问答）基准数据集的很大比例，而这类问题的结构化查询相对比较简单，只涉及一个<主语、谓语、宾语>三元组。因此，不需要一个结构预测过程来恢复多个实体和关系之间固有的查询结构。

其实这里的关键思想与 Bordes 等人提出的思想非常类似，即自然语言问题中表达的关系模式和结构化知识库中的关系/谓词可以通过 CNN 投射到相同低维度的语义空间中。同样，知识库中的实体表现形式与问题中提到的实体相同，并且可以由 CNN 捕获到。因此，CNNSM 可以提供自然语言问题和知识库中的候选三元组之间的相似性，并选择得分最高的一个作为最终答案。

这种解决方案得益于卷积神经网络模型，其优于简单的词袋模式，并且在一定程度上以 Letter-Trigrams 向量作为输入来处理词汇表外（Out-Of-Vocabulary，OOV）问题。但是，这也让我们想起了 KBQA 任务中的两个重要问题：实体链接和关系识别，它们本身都具有足够的挑战性，并且需要足够的训练数据，即“分类-实体”对和“自然语言模式-知识库关系”对来训练模型。特别是目前大型知识库中存在大量的实体和关系，如 Freebase，使得处理多个实体和关系的问题变得更加具有挑战性。

另一方面，Dong 等人（2015）提出使用 CNN 对问题和候选答案之间的不同类型特征进行编码。他们提出利用多列卷积神经网络（Multicolumn Convolutional Neural Network，MCCNN）模型来捕捉一个问题的不同方面，并通过三个渠道（答案路径、答案语境和答案类型）进一步评分一对问答。

与简单的向量表示（Bordes 等，2014）相比，MCCNN 使用 CNN 抽取不同的特征，这些特征可以显式地捕获问题中的主题实体与知识库上的候选答案之间的路径以及期望的答案类型，这两点在评估一个候选答案时显得尤为重要。通过向神经网络中添加所需的列，MCCNN 还可以轻松扩展更多类型的功能。

对于基于特征的模型，实体链接仍然是一个悬而未决的问题。应答路径的编码有助于 MCCNN 在某种程度上沿路径执行浅层推理，然而由于典型的检索-嵌入-比较（Retrieval Embedding Comparing）框架的性质，MCCNN 仍然无法找到更好的解决方案来处理候选答案之间的比较。

3. 利用注意力机制嵌入特征

Hao 等人（2017 年）采用了双向 RNN 模型来捕获给定问题的语义。他们认为，一个问题应该根据不同答案不同方面的关注点（答案方面可以是答案实体本身、答案类型、答案语境等）以不同的方式表示。以“谁是百度公司董事长？”为题，并将其候选答案之一“李彦宏”作为一个例子。在处理答案实体“李彦宏”时，问题中的“董事长“和“百度公司”两个词更加集中，问题表征应偏向于这两个词。而当面对答案类型是“公司”或“董事会成员”时，“谁”则应该是最突出的词。同时，有些问题可能更看重答案类型而不是其他方面。在其他问题中，答案的关系可能是我们应该考虑的重要信息，对于不同的问题和答案，它是动态和灵活的。显然，这需要一种注意力机制，它揭示了问题表征与相应答案方面之间的相互影响。

在处理答案不同方面（包括答案路径、答案语境和答案类型）时，Hao 等人提出了一种基于交叉注意力的神经网络来执行 KBQA ，而不是使用具有不同参数的三个 CNN 来表示问题。

交叉注意力模型代表问题和答案方面之间的相互关注，包含两个部分：答案-问题注意部分和问题-答案注意力部分，前者有助于学习灵活、充分的问题表示，后者有助于调整问答权重值。最后计算问题与不同方面的每个对应候选答案之间的相似度得分，并根据相应的问题-答案权重值将每个候选项的最终得分组合在一起，选择得分最高的候选项作为最终答案。

4. 利用记忆机制回答问题

记忆网络是一种新型的学习框架，它是围绕一种可以在特定任务中读取和修改/附加记忆机制而设计的（Weston 等人，2015）。在记忆网络范式下，基于知识库问答任务的研究已经有了一些尝试，主要是遵循信息抽取模式的检索比较方式。

Bordes 等人于 2015 年进行了第一次尝试，主要侧重于简单问题的讨论，问题可以用一个<主体、关系、对象>（<subject、relation、object>）三元组来回答。在输入组件中，我们以 bag-of-symbols（符号袋）的形式读取结构化的知识库并将其存储在记忆中，并且问题被处理成 bag-of-ngrams 的形式。然后在输出组件中，将 bag-of-ngrams 化的问题与记忆中的条目进行比较以找到候选三元组，并用输入的问题进行求值（得分）。其中，得分最高的三元组的对象将由响应组件提供且作为模型给出答案。这应该是 KBQA 任务中记忆网络的一种简单应用，实际上显示了记忆网络在管理大量知识库（KB）条目方面的潜力，甚至在管理来自多个资源的大量知识库条目时也是如此，如图 12-3 所示。

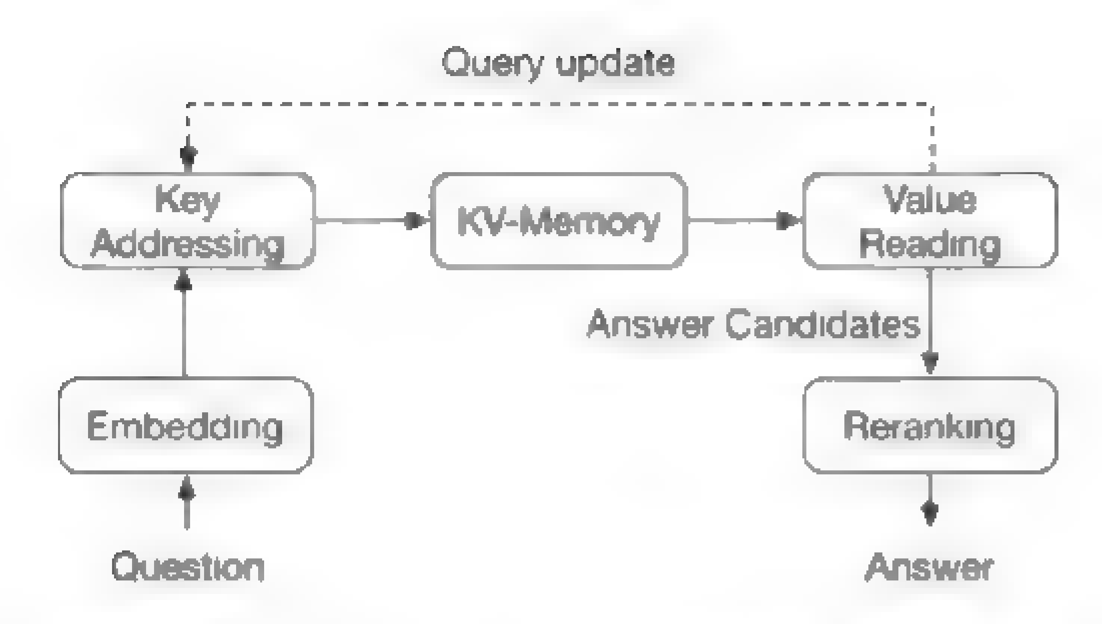


图 12-3　键-值（Key-Value）记忆网络的示意图（Miller 等，2016）

米勒等人（Miller，2016）通过研究记忆机制中各种形式的键-值（Key-Value）的知识库进一步扩展了这一思想，改进后的模型还允许从记忆中读取多个地址来收集线索/上下文，从而动态更新问题以获得最终答案。键-值（Key-Value）设计的优点是使记忆机制更加灵活地存储各种知识，从知识库三元组（主体+关系作为键 Key，对象作为值 Value）到文档（句子或单词窗口作为键或值），它支持用异构资源来回答更复杂的问题。

12.2.2　语义分析模式

语义分析的思路是通过对自然语言进行语义上的分析，转化成为一种能够让知识库“看懂”的语义表示，进而对知识库中的知识进行推理（Inference）、查询（Query）得出最终的答案。简而言之，语义解析要做的事情就是将自然语言的问题转化为一种能够让知识库“看懂”的语义表示，即逻辑形式（Logic Form）。

“检索-嵌入-比较”框架其实是受益于各种神经网络组件来捕获问答相似度的，并且在简单的问题中表现得更好，其中的实体和关系位于知识库中简单化的子图中。但是，它们并不善于解决复杂的语义组合，因为在理解问题时并没有明确的信息抽取机制来捕捉这种组合。相比之下，KBQA 中的其他主流工作（语义分析模式模型）在尝试正式地表示问题的含义，然后使用知识库进行实例化以构建基于知识库的结构化查询，从而可以显式地捕获复杂的查询。

因此，这些模型的核心组件是从自然语言问题中恢复形式意义的表示，如逻辑形式或结构化查询，并通过将表示映射到知识库组件并查询知识库，从而在知识库中找到答案。深度学习方法的工作主要是为了改进框架的某些组件。

也可以将 12.2.1 小节中讨论的 CNNSM 模型（Yih 等，2014）看作是一种语义分析模式的方法，它只能产生一个<subject、predicate、object>三元组作为查询，其中 CNN 用于执行实体链接和关系识别，但不适合用于稍微复杂的问题，如涉及多个实体和关系，以及约束限制方面。主要是因为神经网络组件只负责与知识库组件（实体或关系）之间的映射，而没有明确的机制来识别多个实体或关系之间的内在结构。事实上，在传统的基于语义分析模型中，已经通过 PCCG、PCFG、依赖结构或其他句法、语义分析范式对这些结构进行了深入研究。

1. STAGG：搜索和剪枝时的语义分析

除了单一关系问题，Yih 等人（2015）建议使用查询图的方式来表示问题的含义，其中包含 4 种节点：基础实体（Grounded Entity，标准实体）、存在变量（Existential Variable）、λ 变量和约束/函数。在这里，λ 变量是非标准的实体，并且有望成为最终的答案；存在变量可以指代（引用）中间节点，如“小明的父亲的母亲”中的“父亲”的表达，或者抽象节点，如 Freebase 中的复合值类型（CVT）节点，并且约束或函数被设计为根据某些数值属性（如 argmin）过滤一组实体。在查询图中，节点之间通过有向边连接，表示两个节点之间的关系期望通过知识库谓词进行映射，如图 12-4 所示。

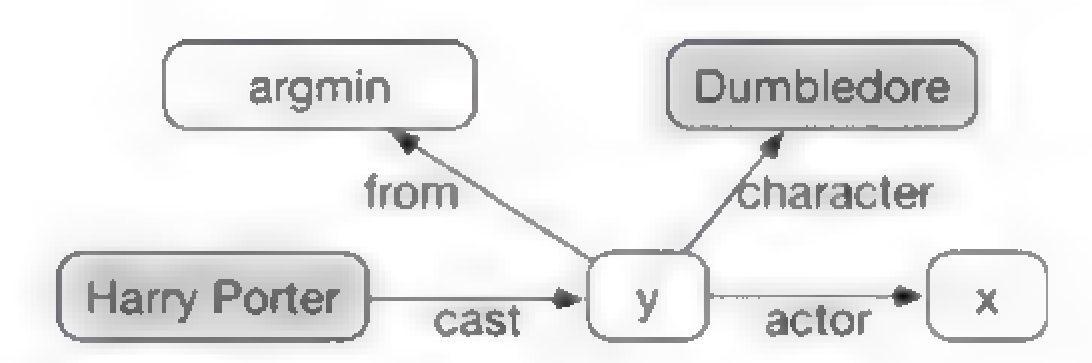


图 12-4　Yih 等人（2015）提出的分阶段查询图生成模型（STAGG）

然后，任务变成如何将一个自然语言问题转换成这样的查询图。Yih 等人（2015）提出了一个分阶段查询图生成模型（STAGG），利用知识库从头逐步剪枝搜索空间，并构建结构化查询。

STAGG 的关键组件包括主题实体链接和识别核心推理链，最后用约束和函数进行扩充，这基本上是一个逐步搜索的解析和排序过程。这里核心推理链捕获主题实体和 lambda 变量之间的关系，并为查询提供主干支撑。Yih 等人（2015）使用深度卷积神经网络模型在语义上匹配一个问题和一个谓词序列（长度为 2，中间有一个 CVT 节点）。

尽管 STAGG 在基准数据集上取得了成功，但是我们可以从 STAGG 的设计中吸取一些经验。

主题实体：第一步，找到主题实体并链接到知识库，也是关键的一步。STAGG 使用了 S-MART（Yang 和 Chang，2015），这是一种短文本实体链接的统计模型，它对后续步骤及整体性能都起着重要作用。当将主题实体链接更改为 Freebase API 时，STAGG 的 F1 总分下降 4.1%。

识别核心推理链：基本上这是一个关系抽取步骤，用于捕获如何从 KB 图上的主题实体开始获取 λ 变量，通过 CNN 捕获，类似于 Yih 等人 2014 年提出的 CNNSM。鉴于所有候选关系的巨大空间，STAGG 仅考虑与主题实体相关的那些候选项，并且捕获问题如何在语义上与主题实体周围的 KB 关系序列匹配。因此，识别核心推理链的这一过程成为匹配和排名（Match-and-Rank）的步骤，同时避免了大规模的多分类方式。

增加约束和聚合：STAGG 会将问题中的其他实体或时间表达式看成为核心推理链的约束节点，并且还会引入某些函数以进一步过滤答案，如将 first、smallest 转换为 arg min。通过一组规则，我们将看到 KBQA 系统会引入聚合函数作为正式表示的一部分。

理解最高级别的表达方式：正如 Berant 和 Liang（2014）、Zhang 等（2015）中所讨论的那样，在问题中可以看到最高级的话语（表达）。大多数 KBQA 工作都采用模板或规则来分析最高级的表达式，只需从 arg min 或 arg max 中进行选择即可（Berant 和 Liang，2014；Yih 等，2015）。然而，正式将最高级表达式分析作为针对 KB 的结构化比较结构，将有助于 KBQA 系统更好地处理最高级的话语，以及那些具有序数约束的话语。Zhang 等人（2015）设计了一个神经网络模型来

学习最高级别话语和 KB 关系之间的潜在对应关系，作为比较结构中的比较维度。例如，从 the longest river 到元组<river.length、descending、1>，我们期望所有 river 在 KB 谓词 river.length 上进行比较（按降序排序），排名最高的就是我们的目标。

2. 改善关系识别

正如语义分析模式工作中（Kwiatkowski 等，2013；Berant 等，2013；Berant 和 Liang，2014）所讨论的那样，从问题中识别 KB 关系/谓词是成功的关键，其中传统的基于特征的模型难以捕获到句子与 KB（知识库）关系之间的不匹配以及自然语言话语之间的差异。其实，已经有人多次使用 CNN 或 RNN 模型在探索词汇或句法特征的关系提取中尝试应用了深度学习方法很多次（Zeng 等，2014；Liu 等，2015；Xu 等，2015）。

KBQA 中的关系提取组件旨在处理短的语境中基于 KB 的关系，其中有多达数千个候选项。一种可能的解决方案是通过 CNN 在自然语言话语和 KB 关系之间执行语义匹配（Yih 等，2014 和 2015），避免对数百种关系进行直接分类，如图 12-5 所示。

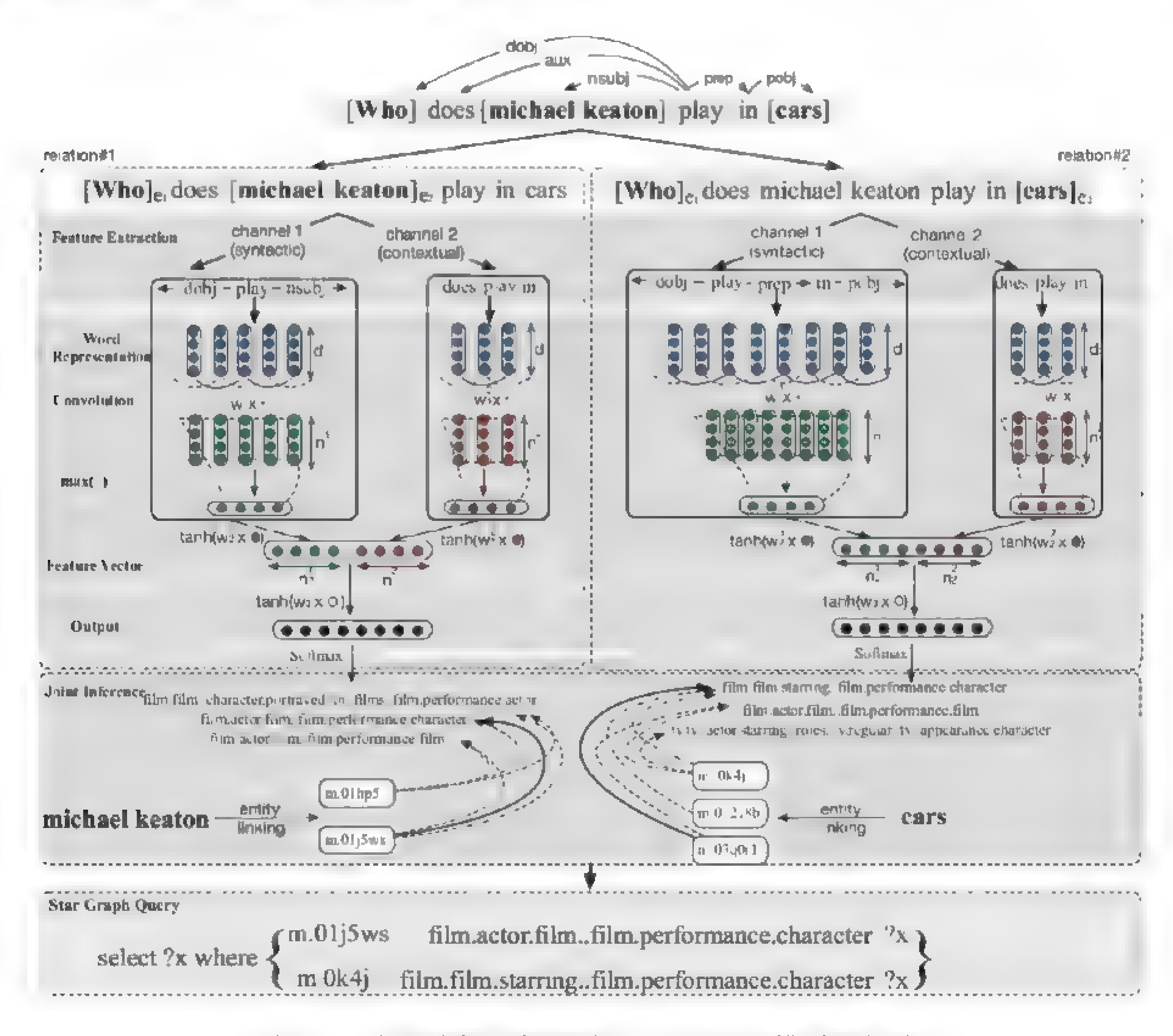


图 12-5　多通道卷积神经网络（MCCNN）模型示意图

Xu 等人（2016）提出了一种多通道卷积神经网络（MCCNN）模型，从词法和句法两方面学习紧凑性、稳健的关系表示，这些方法更适合开放域的 KBQA 场景。因为在一个开放域的 KB 中，通常有上千个关系，传统的基于特征的模型不可避免地会遇到数据稀疏问题和对未知单词泛化能力

差的问题。

3. 神经符号机器

还有另一个有趣的语义分析模式的工作，试图结合神经网络和符号推理的优点来改善问答效果。Liang 等人于 2017 年引入神经符号机器（Neural Symbolic Machine，NSM），其配备了一个神经网络组件，负责从自然语言表示映射到可执行的代码，以及一个符号组件来执行代码，以剪枝搜索空间来找到答案。

具体来说，神经网络组件基本上是一个序列到序列模型，它维护一个关键变量存储器（Key-Variable Memory），在生成程序序列时处理中间结果。但是，神经网络组件和符号解释器的混合设计会使整个框架难以训练，然后要将其转换为一个强化学习问题来解决。

12.2.3 信息抽取与语义分析小结

综上所述，其实很容易发现我们不需要对信息抽取方法和语义分析方法进行明确的区分。这两种观点确实各有优势，信息抽取（IE）方法更多地利用了新型神经网络模型和架构的优点，以便在压缩的语义空间中更好地表示问题和候选答案，并且更容易在模型结构中合并各种特征表示。另一方面，深度学习为语义分析（SP）模型提供了更精确的关系或约束识别/映射，并支持更精确/复杂的含义表示和推导。

事实上，之前很多的成果都可以被认为是具有这两方面的特性，尤其是那些针对简单问题或受益于这两种模式的成果。例如，STAGG 遵循传统的语义分析模式，从一个问题构造结构化查询，其分级-剪枝有助于剪枝搜索空间，从而获得更好的查询结构和整体性能。我们认为，一些具有信息抽取和语义分析两种模式优点的新范式，如记忆网络模型和神经符号框架，具有足够的灵活性，可以适用于更复杂的问题。

12.2.4 挑战

随着基于知识库问答系统的发展，已经出现了几个最受关注或讨论的问题，尤其是在使用深度学习模型的背景下。

1. 组合性

传统基于语义分析的 KBQA（基于知识库的问答）工作通常依赖组合分类语法（CCG）或概率从问题中推导出其意义表示（Cai 和 Yates，2013；Kwiatkowski 等，2013），如果不考虑这些句法结构，如信息抽取模式的方法，那么在统一模型中进行显式捕获是相对困难的。因此，许多现有的工作（成果）依赖于手工定义的规则或模板来处理组合性（Yih 等，2015）。

2. 自然语言与知识库之间的差距

我们已经讨论过，当尝试检索候选答案或将自然语言话语与知识库项目相匹配时，实体链接和关系抽取是两个主要障碍。其主要原因是自然语言与知识库的不匹配，包括语言方面上下文的限制

或省略、了词汇的构词性，甚至是知识库设计的缺陷。目前，相关研究人员已经提出了各种神经网络模型来改进关系匹配或抽取方法，但对实体链接任务的关注却远远不够，而实体链接任务又是KBQA 系统的基本步骤。

3. 训练数据

关于训练数据方面，其实训练数据一直是各种机器学习方法中存在的问题，尤其是神经网络模型，它比传统方法需要更多的训练数据。在问答场景中，收集问答配对的成本是非常昂贵的，更不用说任何好的注释，如逻辑形式、结构化查询，甚至实体和关系注释。可能的解决方案包括使用问答配对作为间接监督来收集伪标签（Yih 等，2015；Xu 等，2016），或者使用噪声标签或模板自动收集训练数据（Miller 等，2016；Bordes 等，2014a）。

KBQA（基于知识库的问答）是一项具有挑战性的任务，它需要许多 NLP 或 IR 技术，如词汇分析、句法分析、信息抽取、实体链接、推理等，而深度学习的最新进展为提高 KBQA 提供了有用的工具或新的框架，这些在早期是大家公认的。我们相信，神经网络建模与问答的深度融合将会给这一领域带来更多的机遇。

12.3 机器理解中的深度学习

12.3.1 任务描述

机器理解（Machine Comprehension，MC）是近年来在自然语言处理和人工智能领域得到广泛应用的一种新的应用。机器理解具备测试机器阅读、处理和理解文本的能力。机器理解遵循传统的 QA（问答）设置，但仍存在一些差异。

（1）在传统的 QA 中，对于给定的问题，答案的来源可能各种各样，比较繁杂，如基于知识库的问答（KBQA）、Web 搜索结果，甚至一些问答平台（也称为社区 QA）。而在机器理解中，上下文的知识则仅限于给定的单个文档。

（2）与传统的 QA 相比，特别是对于 IRQA 和 KBQA，机器理解主要关注那些无法直接回答的问题，需要根据给定文档中的多个实体或事件进行推理。另外，机器理解中需要推理能力。

（3）与传统的 QA 相比，机器理解中的答案类型更加多样化，从单个单词到多个句子不等。此外，机器理解的问题形式也是多种多样的，如多项选择题（前面提供的答案候选项）和完形填空式问题（没有提供候选项，需要从系统中生成答案）。

1. 数据集

MCTest：机器理解的任务始于自然语言处理（NLP）社区。2013 年，微软研究人员提出了 MCTest（Richardson 等，2013）数据集来评估机器的理解能力。在 MCTest 中，每个文档（故事）都与 4 个问题相关联。对于每个问题，提供 4 个候选答案，并且系统需要选择正确的答案。MCTest 的一个例子如图 12-6 所示。

Alyssa got to the beach after a long trip. She's from Charlotte She traveled from Atlanta She' s now in Miami. She went to Miami to visit some friends

The girls went to a restaurant for dinner. The restaurant had a special on catfish. Alyssa enjoyed the restaurant's special. Ellen ordered a salad. Kristin had soup. Rachel had a steak.

1: one: Why did Alyssa go to Miami?
A) swim
B) travel
*C) visit friends
D) laying out

2: multiple. What did Alyssa eat at the restaurant?
A) steak
B) soup
C) salad
*D) catfish

图 12-6 MCTest 数据中故事和问题的示例

显然，MCTest 是一个标准的阅读理解数据集，其中的故事是虚构的，一些问题可以从几个句子（标记为多个）中进行回答。作者将数据集分为两个子集，包括 MC160 和 MC500，分别包含 160 和 500 个故事，但是此数据集的规模太小，有时仅用作测试设置。近年来，许多研究者通常借助外部语言工具来提取特征，然后在此基础上进行推理。从 MCTest 开始，已经发布了几个机器理解数据集。在这里，我们主要介绍以下几个标准的数据资源。

bAbi：bAbi（Weston 等，2015）是一个机器理解数据集，根据作者的描述是人工智能完全问题（自然语言处理被认为是人工智能完全问题（AI Complete），也就是说，如果自然语言处理实现了，人工智能也就实现了）。总的来说，bAbi 包含 20 个子任务，其中每个子任务需要不同的应答技巧。一些子任务示例如图 12-7 所示。

John is in the playground.
Bob is in the office.
Where is John? *playground*

single supporting fact

John is in the playground.
Daniel picks up the milk.
Is John in the classroom? *no*
Does Daniel have the milk? *yes*

Yes or No questions

John is in the playground.
Bob is in the office.
John picked up the football.
Bob went to the kitchen.
Where is the football? *playground*
Where was Bob before the kitchen? *office*

two supporting facts

The office is north of the bedroom.
The bedroom is north of the bathroom.
What is north of the bedroom? *office*
What is the bedroom north of? *bathroom*

Subject vs. Object

图 12-7 bAbi 数据集中部分子任务

由于该数据集被划分为不同的类别，因此不同子任务上的性能可能会暴露出一个模型在不同问题类型上的优缺点。整个数据集是由几个人工设计的规则自动合成和自动生成的，虽然规则应该是无限制的，但实际上生成规则仅基于不超过 100 个单词。因此，此数据集中的某些问题或文档是重复的。由于 bAbi 是由规则自动合成的，因此被利用的算法或系统更可能接近所使用的生成规则。

SQuAD：SQuAD（Rajpurkar 等，2016）是斯坦福问答数据集，是最近发布的一个人工创建的大型机器理解数据集。该数据集包含近 100 000 个“文档 问题”配对，这些文档来源于维基百科

页面，然后注释者根据这些文档提出一些问题，并在文档中标注出相应的答案。请注意，在 SQuAD 中没有提供候选答案。系统可以通过预测答案在文档的开始和结束位置来“生成”答案。这里的问题示例如图 12-8 所示。

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain in scattered locations are called "showers".

Where do water droplets collide with ice crystals to form precipitation?

图 12-8　SQuAD 数据集中问题示例

此外，最近还发布了几个与 SQuAD 规模和形式都相似的机器理解数据集，如 NewsQA 7 和 Marco。

完形填空式机器理解数据集：除了机器理解中提到的 QA 形式之外，完形填空式查询（Taylor，1953）也是其中的基本形式之一。这种类型具有阅读理解的大部分特征，但答案只是文档中的一个单词。实际上，最近提出了许多这类的数据集，如 CNN/Daily Mail（Hermann 等，2015）和 CBT（Hill 等，2015）。在 CNN/Daily Mail 中，作者提出了一种从两个新闻语料库中自动生成完形填空的方法，每条新闻报道都有一个标题或摘要。如果作者删除标题中的一个具体名词，系统就需要根据给定的文档来填充这个占位符。为了避免语言建模或超出文本理解范围的现实世界知识的影响，作者对文档和查询中涉及的所有实体进行了匿名化处理。在 CBT 中，每个文档包含 20 个连续的句子，第 21 句中有一个词被移去了。为了避免在阅读理解中使用基于语言建模的方法，答案仅限于专有名词。图 12-9 给出了 CNN/Daily Mail 的一个例子。

CONTEXT:
(@entity4) if you feel a ripple in the force today , it may be the news that the official @entity6 is getting its first gay character . according to the sci-fi website @entity9 , the upcoming novel " @entity11 " will feature a capable but flawed @entity13 official named @entity14 who " also happens to be a lesbian . " comics and books approved by @entity6 franchise owner @entity22 -- according to @entity24 , editor of " @entity6 " books at @entity28 imprint @entity26 .

QUESTION:
characters in " @placeholder " movies have gradually become more diverse
ANSWER:
@entity6

图 12-9　CNN/Daily Mail 数据集的示例

2. 实现机器理解的知识需求

机器理解是一项综合性的推理任务，需要对自然语言有深入地理解。在心理学中，理解来自于词语之间的相互作用，这些词语可以触发给定段落、文档内部和外部的知识。它是一个创造性的、多方面的过程，依赖于 4 种语言技能：语音学、语法学、语义学和语用学。对于机器理解问题，要实现真正的理解甚至还需要理解多个子句（从句）之间的关系。例如，要理解事件之间的时间关系，隐式地需要识别连接词（when, as, since，…）、时间索引词（morning, evening，…）、时态和方

位（Aspect，笔者认为也可以是角度）（went, is going, will go，…）等表达式。此外，还需要其他推理技巧，例如数学运算需要回答与算术问题有关的问题，如“汤姆有 4 支铅笔，给了同桌 2 支，他还剩几支铅笔？”，需要回答这类问题的系统应该推断出等式“4-2 2”。Sugawara 等人（2017）提出了机器理解需要的 10 个技能，如图 12-10 所示。

Skills	Descriptions or examples
List/Enumeration	Tracking, retaining, and list/enumeration of entities or states
Mathematical operations	Four arithmetic operations and geometric comprehension
Coreference resolution	Detection and resolution of coreference
Logical reasoning	Induction, deduction, conditional statement, and quantifier
Analogy	Trope in figures of speech, e.g., metaphor
Spatiotemporal relations	Spatial and/or temporal relations of events
Causal relations	Relations of events expressed by why, because, the reason for, and so on
Commonsense reasoning	Taxonomic knowledge, qualitative knowledge, action and event changes
Schematic/Rhetorical clause relations	Coordination or subordination of clauses in a sentence
Special sentence structure	Scheme in figures of speech, constructions, and punctuation marks in a sentence

图 12-10 机器理解需要的技能

一般来说，机器理解涉及处理许多语言模式（Pattern），如词汇、句法或高级语篇、释义。为了对这些特征进行建模，根据方法论的观点，可以将当前的方法分为两部分：基于特征工程的方法和基于深度学习的方法，接下来我们对此将做一些阐述。

12.3.2 基于特征工程的机器理解方法

目前，现有的基于特征工程的方法通常是将文本理解任务建模为计算给定问题与文档或段落之间语义相似性的任务。这些方法试图通过基于 POS 标记的特征、依存分析特征、共指、引用等浅层语言学特征对句子和文档的语义进行建模。基于不同的特征，可以捕获不同类型的语义，如词汇层次语义、语篇（有时也叫话语）的语义等。

1. 词汇匹配

词汇匹配是机器理解任务中一种简单而有效的方法。这种方法通常采用一种基于滑动窗口的算法，通过为每个与问题文本配对的答案形成词袋向量，并对这些候选答案进行排名（Rank）。然后根据每个候选项与故事文本的重叠程度对它们进行评分，并计算出得分最高的候选项。具体来说，这个算法在整个故事文本上滑动一个窗口，该窗口的大小等于问答配对中的单词个数。故事文本窗口和问答配对之间的最高重叠分数将被作为答案的对应分数。

史密斯等人（2015）通过多次取舍，并对获得的分数求和来对每个答案进行评分。具体而言，它们从窗口大小 2 开始并将其增加到 30（即 30 个切分词），然后将这些得分与整个故事中“问题 答案”配对的总匹配数量结合起来。正如他们宣称的那样，这种解决方案可以让系统捕捉到故事中的远距离关系。MCTest 上原有的和增强的滑动窗口词法匹配方法的比较结果如表 12-1 所示。

表 12-1 MCTest 上词汇匹配方法的表现情况

	滑动窗口 Sliding window(%)	增强型滑动窗口 Enhanced sliding window(%)
MC160	69.43	72.65
MC500	63.01	63.57

2. 话语（Discourse，语篇）关系

回答一个问题所需的相关信息可以分布在多个句子中，理解这些句子之间的语义关系对于找到正确答案是很重要的。以图 12-11 为例，为了回答“为什么莎莉穿上她的鞋子”的问题，我们需要推断出“她穿上鞋子”和“她外出散步”之间存在因果关系。

Sally liked going outside. She put on her shoes. She went outside to walk. [...] Missy the cat meowed to Sally. Sally waved to Missy the cat [...] Sally hears her name. "Sally, Sally, come home", Sally's mom calls out. Sally runs home to her Mom. Sally liked going outside.

Why did Sally put on her shoes?

A) To wave to Missy the cat
B) To hear her name
C) Because she wanted to go outside
D) To come home

图 12-11 一个需要多句推理的问题示例

实际上，一些先前的研究成果已经证明了话语关系在一些领域应用中存在的价值，如问答领域（Jansen 等，2014）。Narasimhan 和 Barzilay 于 2015 年提出了三种模型，将话语关系纳入了机器理解系统中。

将文档中的句子表示为 z，问题表示为 q，答案表示为 a。

模型 1：

$$P(a,z\mid q_j)=P(z\mid q_j)P(a\mid z,q_j) \tag{12.1}$$

在公式（12.1）中，我们将联合概率定义为两个分布（有的称为两个分量概率）的乘积。其中，第一分布是问题中给定段落里句子的条件分布，这是为了帮助识别出回答问题所需的句子；第二个分布是通过给定问题 q 和句子 z 来对选择答案的条件概率进行建模。我们可以使用指数族来得到这些分量概率，即$P(z\mid q)\propto \exp^{(\theta_1\phi_1(q,z))}$和$P(z\mid a,q)\propto \exp^{(\theta_2\phi_2(q,a,z))}$，其中$\phi$是特征向量，θ 代表关联权重值（Associate Weight）。对文档中所有句子z_n求和，可以得到具体答案的概率a_j：

$$P(a_j\mid q_j)=\textstyle\sum_n P(a_j,z_n\mid q_j) \tag{12.2}$$

通过这种方式可以将目标似然函数（Likelihood Objective Function）写成：

$$L_1(\theta)=\log\textstyle\sum_j\sum_n P\,(a_j,z_n\mid q_j) \tag{12.3}$$

模型 2：

上述模型只能一次考虑一个支持句子（即 z），当然，也可以把它扩展到多个句子。我们对给定的问题使用多个相关句子，在这种情况下，联合模型的定义如下：

$$P(a,z_1,z_2 \mid q)=P(z_1 \mid q)P(z_2 \mid z_1,q)P(a \mid z_1,z_2,q) \tag{12.4}$$

给定问题 q，我们首先预测与概率 P（z_1| q）相关的第一个句子支持句 z_1，然后给出 q 和 z_1，接着推理出第二个句子支持句 z_2，最后答案 a 便是模型预测出的结果。

模型 3：

该模型试图直接指定问题之间的话语关系，然后利用这种关系对文档中的其他相关句子进行推理。具体来说，模型 3 添加了一个隐藏变量 r∈R 来表示关系类型，它集成了将问题类型与关系类型联系在一起的特性，还利用关系类型来计算句子之间的词汇和句法相似性。

关系集 R 由以下关系组成。

（1）因果关系：事件的原因或事实的原因。

（2）时态：事件的时间顺序。

（3）说明：主要处理 how-type 的问题。

（4）其他：上述三种关系以外的关系（包括非关系 non-relation）。

现在，我们对等式（12.4）中的联合概率进行修改，加入关系类型 r，结果如下：

$$P(a,r,z_1,z_2 \mid q)=P(z_1 \mid q)P(r \mid q)P(z_2 \mid z_1,r,q)P(a \mid z_1,z_2,r,q) \tag{12.5}$$

其中，新增的分量（Component）$P(r|q)$是取决于问题的关系类型 r 的条件概率。因此，该模型可以学习诸如对应于因果关系的 why-questions。下面我们给出这三种模型的结果情况，如表 12-2 所示。

表 12-2 MCTest 上三种模型的准确性情况

	MC160			MC150		
	Single(%)	Multi(%)	All(%)	Single(%)	Multi(%)	All(%)
Model 1	78.45	60 07	68.47	70.58	57.77	63.58
Model 2	74.68	60 07	66.52	66.17	59.9	62.75
Model 3	72.79	60 07	65.69	68.38	59.9	63.75

其中，Single 是指只需要一个支持句子就能回答的问题；Multi 是指需要多个句子才能回答的问题。

3. 蕴涵答案（Answer-Entailing）结构

以前的一些 NLP 研究成果得益于学习两个文本片段之间的潜在结构。例如，在识别文本蕴涵（RTE）时，可以通过它们之间的某些潜在对齐从其前提中推断出该假设。在机器理解中，我们还可以将这种蕴涵（Entailing）结构信息纳入其中。在图 12-6 所示的例子中，为了回答第二个问题，我们可以使用一些句法规则将问题和一个候选答案转换成语句。例如，候选答案之一是鲶鱼

（catfish），我们将其与相关查询组合成一个陈述语句：Alyssa 在餐厅吃鲶鱼（catfish）。把这个陈述作为一个假设，文档作为前提，我们可以推断出这个蕴涵（Entailment）的概率。此时，我们可以给出该例推理过程的示意图，如图 12-12 所示。

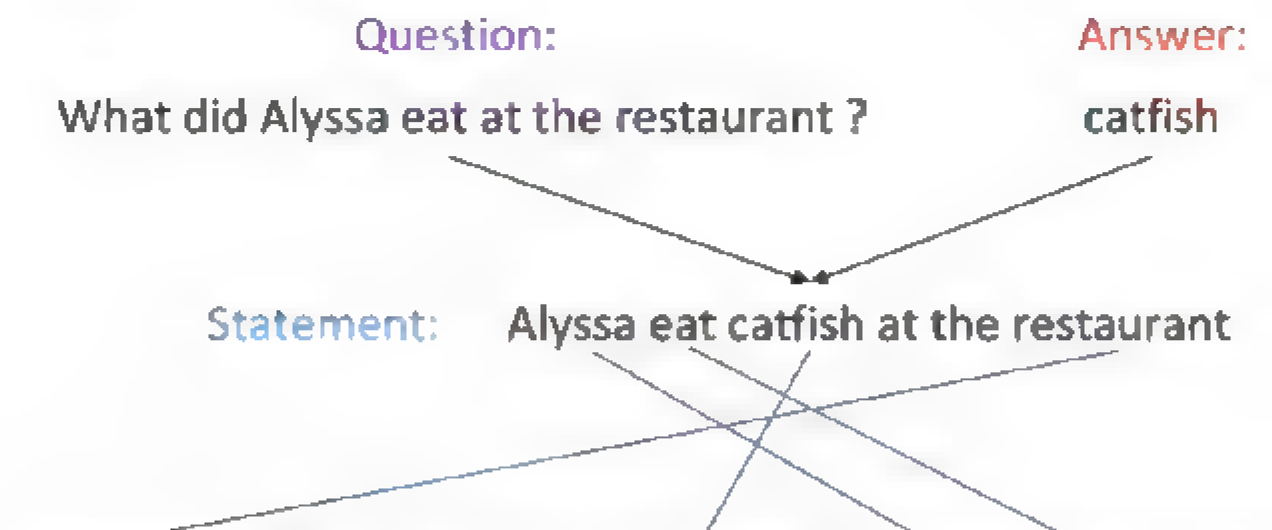


图 12-12　MCTest 里一个蕴涵答案结构的示例图

这里我们考虑的蕴涵答案结构可以将文本中的多个句子与假设对齐，并且对齐文本中的句子并不限于在文本中连续出现。为了实现这种不连续的对齐，Sachan 等人（2015）利用了文档结构，尤其是他们借助于修辞结构理论，可以捕捉到事件或实体在句子之间的共指联系。他们使用潜在结构 SVM（LSSVM）专门训练了一种最大边际方式（Max-Margin Fashion），其中蕴涵答案的结构是潜在的。MC500 上这种蕴涵答案模型的实验结果如表 12-3 所示。

表 12-3　MC500 上蕴涵答案模型的准确率

	Single	Multiple	All
Accuracy(%)	67.65	67.99	67.83

4. 基于特征工程方法所面临的挑战

基于特征工程的方法是处理机器理解问题的一种有效而明确的方法，它们通常利用几种语言特征来建模对给定文档和问题之间的语义关系进行建模，然后根据这些特征进行推理，该过程清晰且易于理解该方法中出现的问题，但是这些语言特征有时需要通过经验或启发式实验来获得。另外，它们严重依赖于独立的语言工具，如词性标注、语法分析器等，这可能会给系统带来噪声。因此，特征工程方法通常以 MCTest 等机器理解数据为研究对象，而对于一些大规模的机器理解数据集，如 SQuAD 和 bAbi，现有的基于特征工程的方法很难从文本中进行设计并提取出有效的特征。近年来，随着深度学习在计算机视觉和语音识别方面取得的巨大成功，越来越多的研究人员开始关注基于深度学习的机器理解技术。

12.3.3　机器理解中的深度学习方法

对于机器理解任务，我们将介绍几种流行的基于不同数据集的深度学习方法。现在给定一个文档 d 和一个问题 q，对选择答案 a 的概率情况可以进行如下建模：

$$P(a \mid d,q) \propto \exp(W(a)g(d,q)) \tag{12.6}$$

其中，W(a)表示答案候选项 a 的嵌入，g(d,q)表示给定问题 q 下文档 d 的嵌入。关键的部分是计算函数 g(d,q)，可以应用若干深度神经网络，如 RNN、LSTM 和记忆网络（Weston 等，2015b）。

1. 基于 LSTM 的编码器

长短期记忆网络（LSTM）已被证明能够有效地将序列数据建模为向量。因此，为了对函数 g(d,q)进行建模，Hermann 等人（2015）将相关文档一次性输入到基于 LSTM 的编码器中。问题 q 也在分隔符之后被提供给该编码器。这样给定文档 d 和问题 q 的配对可以是一个很长的单序列，如图 12-13 所示。

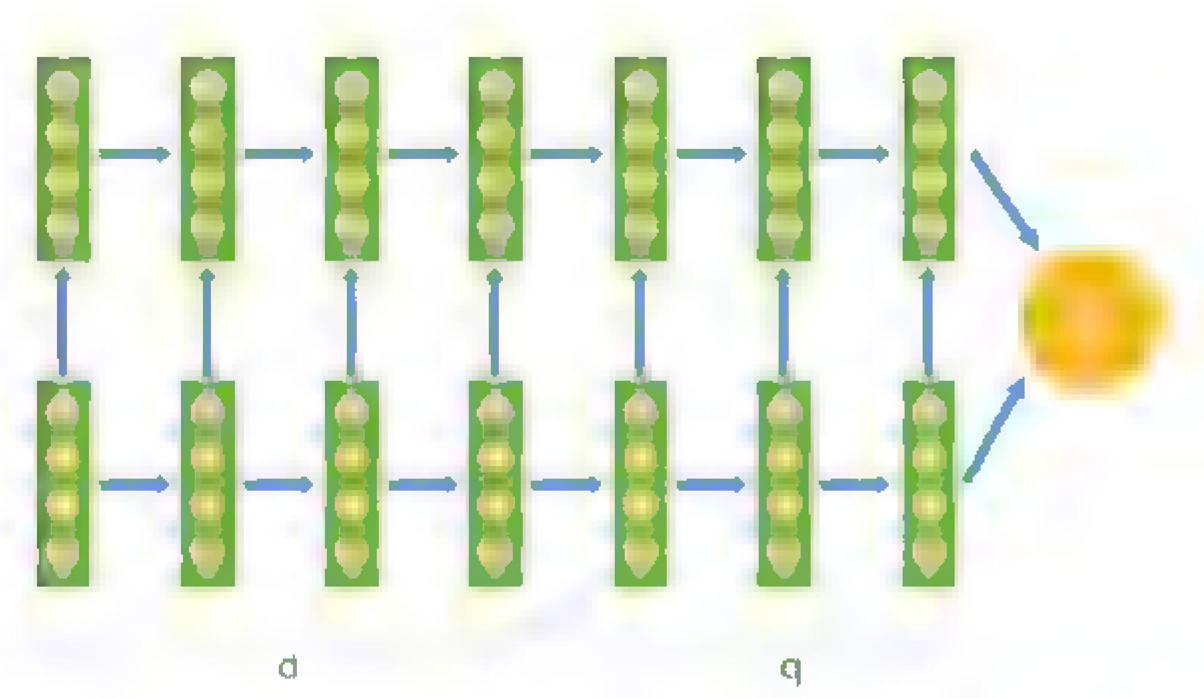


图 12-13　给定文档 d 和问题 q 的配对可以是一个很长的单序列

2. 双向注意力编码器

因为单向 LSTM 很难在长距离上传播依赖关系，所以信息在从一个组件到另一个组件的传输过程中会慢慢衰减，进而使得文档的语义不能被准确地编码。因此，越来越多的研究人员采用双向 LSTM 模型对序列数据进行编码。另外，文档 d 中并非所有的句子或上下文都与给定的问题 q 有关联。例如，这里有一个文档 d，其内容是“迈克尔 • 乔丹在 1993-94 赛季开始之前突然从芝加哥公牛队退役，去追求自己的棒球职业生涯”。当问题 q 是“迈克尔 • 乔丹何时从 NBA 退役的？”时，文档 d 中的关注点应该是“在 1993-94 赛季开始之前”。当问题 q 是“迈克尔乔丹从 NBA 退役后参加了哪项运动？”时，文档 d 中的关注点应该是“追求棒球生涯”。也就是说，在处理不同的问题时，我们应该对文档 d 中不同部分给予不同的关注。因此，将注意力机制引入深度神经网络也是很自然的事情。Chen 等人（2016）提出了一种具有注意力机制的双向编码模型（BiDEA），该模型在 CNN/Daily Mail 数据集上取得了良好的性能。

关于该模型的结构其实是非常直观的，其预测答案的过程主要包括以下三个步骤。

（1）编码：在所有词被映射到 d 维向量之后，段落 p(d)和查询 q 可以被分别表示为$p_1, p_2, \ldots, p_m$和$q_1, q_2, \ldots, q_l$。因此，段落 p 的上下文信息可以通过下面的公式计算得到。

$$\overrightarrow{h_i} = LSTM(\overrightarrow{h_{i-1}, p_i}), i = 1, \ldots, m$$
$$\overleftarrow{h_i} = LSTM(\overleftarrow{h_{i+1}, p_i}), i = m, \ldots, 1$$
$$\widetilde{p_i} = concat(\overrightarrow{h_i}, \overleftarrow{h_i}).$$

同时，问题可以以相同的方式由另一个 LSTM 层嵌入到 q（单个向量）中。

（2）注意力：p_i中的所有文本信息可以通过以下方式组合进输出向量O中。

$$\alpha_i = softmax_i q^T W_S \widetilde{p_i}$$

$$O = \sum_i \alpha_i p_i$$

在上述等式中，$W_S \in \mathbb{R}^{(h \times h)}$用于测量问题 q 与段落 p_i 中单词之间的相似性。

（3）预测：预测答案 a 的计算方法如下。

$$a = argmax_{(a \epsilon p \cap E)} W_a^T O$$

其中，E 是嵌入矩阵，W_a是输出O和候选词 a 之间的测量矩阵。

虽然上述模型的计算非常简单，但是在数据集 CNN/Daily Mail 上却获得了非常好的性能表现（实验结果如表 12-4 所示）。根据 Chen 等人（2016）的分析，其提出的模型的有效性是由于：①CNN/Daily Mail 的推理水平仍然很简单，可以用一个简单的模型来处理；②各类模型在 CNN/Daily Mail 上已经达到了性能上限，甚至可以通过信息检索系统对该语料库很好地做处理。

表 12-4 BiDEA 等模型在 CNN/Daily mail 上的实验结果

	CNN		Daily mail	
	Val	Test	Val	Test
Attentive reader（Hermann 等，2015）	61.6	63.0	70.5	69.0
MemNN（Sukhbaatar 等，2015）	63.4	6.8	—	—
AS reader（Hermann 等，2015）	68.6	69.5	75.0	73.9
Stanford AR（Chen 等，2016）	68.6	69.5	75.0	73.9
DER network（Kobayashi 等，2016）	71.3	72.9	—	—
Iterative attention（Sordoni 等，2016）	72.6	73.3	—	—
EpiReader（Trischler 等，2016）	73.4	74.0	—	—
GAReader（Dhingra 等，2016）	73.0	73.8	76 7	75.7
AoA reader（Cui 等，2017）	73.1	74.4	—	—
ReasoNet（Shen 等，2017）	72.9	74.7	77 6	76.6
BiDAF（Seo 等，2016）	76.3	76.9	80 3	79.6
BiDEA（Chen 等，2016）	72.4	72.4	76.9	75.8

此外，为了以不同颗粒度表示上下文并实现查询感知的上下文表示，Seo 等人（2016）采用多级递阶过程提出了用于机器理解任务的双向注意力网络（BiDAF）。

如图 12-14 所示，模型主要由以下 6 层组成。

- 字符嵌入层：字符级别的 CNN，可以将单词中的字符映射到连续向量。
- 词嵌入层：预先训练的词向量矩阵。
- 短语嵌入层：双向 LSTM 层，可以捕获单词的上下文信息。
- 注意力层：相似性矩阵 S，测量来自两个方向的上下文和查询词之间的相似性，即上下文到查询和查询到上下文。
- 建模层：包含所有单词上下文信息的两层双向 LSTM。
- 输出层：两个逻辑回归模型，分别捕获起始索引和结束索引。

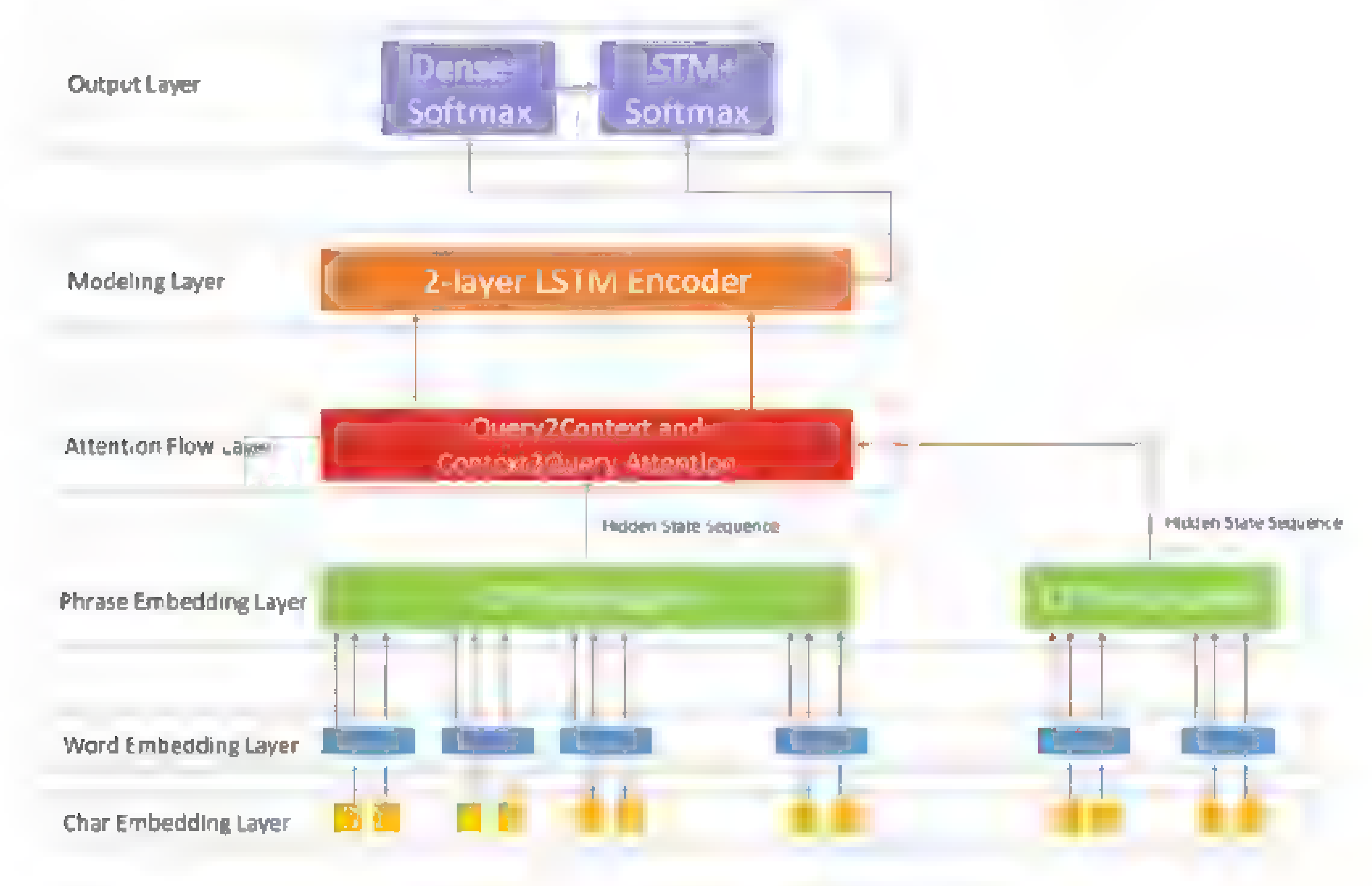


图 12-14 双向注意力模型图

如表 12-5 所示，SQUAD 上的实验结果表明，BiDAF 的方法带来了性能的提高，这可能是由于 BiDAF 在层级中具有发现支持证据的开始点和结束点的能力所引起的。

表 12-5 BiDAF 等模型在 SQuAD 测试集上的实验结果

	Single model		Ensemble	
	EM	F1	EM	F1
Logistic regression baseline（Rajpurkar 等，2016）	40.4	51.0	—	—
Dynamic chunk reader（Yu 等，2016）	62.5	71.0	—	—
Fine-grained gating（Yang 等，2016）	62.5	73.3	—	—
Match LSTM（Wang 和 Jiang，2016）	64.7	73.7	67.9	77.0
Multi-perspective matching（Wang 等，2016）	65.5	75.1	68.2	77.2
Dynamic coattention networks（Xiong 等，2016）	66.2	75.9	71.6	80.4
R-Net（Wang 等，2017）	68.4	77.5	72.1	79.7
BiDAF（Seo 等，2016）	68.0	77.3	73.3	81.1

3. 记忆网络（Memory Network）

Weston 等人于 2015 年提出了记忆网络，用于解决序列神经网络中信息的衰减问题。它还可以用推理组件和长期记忆组件（实际上是一个矩阵或张量，它的名字就是从这里来的）进行推理。总的来说，它包括以下 4 个主要组成部分。

- I（输入特征映射）将输入向量转换为内部特征表示。
- G（泛化）根据新的输入更新现有记忆。
- O（输出特征映射）根据新的输入和当前记忆状态计算新的输出。
- R（响应）将输出转换为所需要的响应格式。

图 12-15 给出了 MemNN（Memory Network）的示意图。记忆网络的一种重要形式是端到端记忆网络，我们将其缩写为 MemN2N。MemN2N 的一个优点是可以以端到端的方式进行训练，这意味着其需要较少的监督信息，并且更普遍地适用于真实场景。以下等式分别是 I、G、O 和 R 中的计算情况：

- I$p_i = Softmax(u_T, m_i)$，其中 m_i =Ax_i（x_i是输入句子的嵌入向量）；$u = Bq$（q 是输入查询）。
- G 在 MemN2N 中，记忆尚未被更新。
- O$o = \sum_i p_i c_i$，其中$c_i = Cx_i$。
- R$\hat{a} = \mathrm{Softmax}(W(o + u))$。

另外，通过以下方式很容易将网络层插入到 MEMN2NS 中。

- 第（k+1）层的 u 可以计算为$u^{(k+1)} = u^k + o^k$。
- 每层都有自己的A^k和C^k。
- 预测可以通过$\hat{a} = \mathrm{Softmax}(Wu^{(k+1)})$来计算。

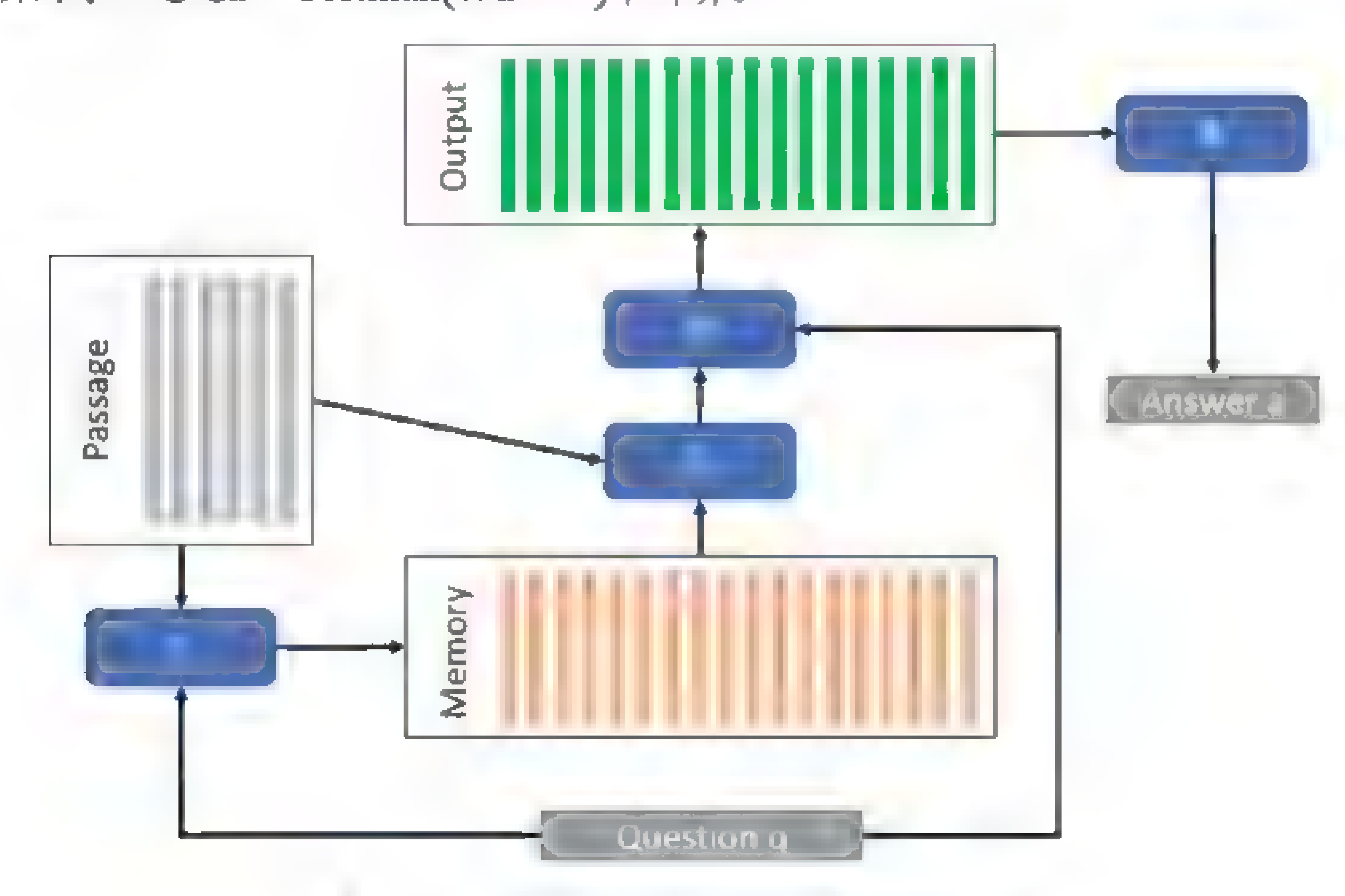


图 12-15　用于 bAbI 任务的记忆网络示意图

MemN2N 最初应用于 bAbI 任务中的 20 个任务。最佳的 MemN2N 性能接近于监督模型，位置编码（PE）表示优于词袋（BoW）模型，线性（LS）训练有助于避免局部极小值，联合训练有助于完成所有任务。

除了记忆神经网络，值得注意的是，G 中的计算实际上是一种注意力机制。而记忆网络是第一个将外部知识保存在指定矩阵中的模型，它对自然语言处理的各种深层模型中的记忆机制的发展有着重要影响。

12.4 利用 TensorFlow 实现问答任务

问答（QA）系统是一个旨在为回答自然语言提出的问题而设计的系统。一些 QA 系统从诸如文本或图像之类的信息源中获取信息以回答具体的问题。这些依赖“信息来源”的系统可以分为两个主要子类别：开放领域，它需要回答的问题几乎可以是里面的任何选项，不限于特定领域；封闭领域，它涉及的问题具有一些特定的限制，因为它们是与一些预定义的信息源相关（如提供的上下文或类似于医学等特定领域）。

我们将创建一个基于神经网络的 QA 系统，并使用封闭领域的信息来源。为了做到这一点，我们使用 Ankit Kumar 等人在其论文《*Ask Me Anything: Dynamic Memory Networks for Natural Language Processing*》中介绍的动态记忆网络（Dynamic Memory Network，DMN）的简化版本（读者也可以自行查阅，笔者在 https://arxiv.org/abs/1506.07285 上获取到该文）。

12.4.1 bAbI 数据集

对于这个项目，我们使用 Facebook 创建的 bAbI 数据集，与所有 QA 数据集一样，此数据集包含了我们需要的所有问题信息。bAbI 数据集中的所有问题都有一个与之相关的上下文，这是一个句子序列，以便能够保证回答问题时存有所必需的详细信息。此外，数据集还为每个问题提供了正确答案。

根据回答问题所需的方法，bAbI 数据集中的问题被划分为 20 个不同的子任务。每个子任务都有自己的一套训练问题集和一套单独的测试问题集，这些子任务测试各种标准的自然语言处理能力，包括时间推理和归纳逻辑。为了更好地了解这一点，我们来看一个 QA 系统将要回答问题的具体示例，如图 12-16 所示。

Context

Fred picked up the apple there. Bill travelled to the kitchen. Bill got the milk there. Jeff went to the kitchen. Bill passed the milk to Jeff. Jeff handed the milk to Bill.

Question	Answer
Who did Jeff give the milk to?	Bill

图 12-16　bAbI 数据集示例图（图片来源：Steven Hewitt）

上图中灰色部分为上下文（或语境），无背景颜色的为问题及其对应答案(斜体为答案)。

此任务测试神经网络对于这三个对象之间内在关系所涉及相关运算或操作的理解。从语法上讲，该任务将测试系统是否能够区分主语、直接宾语和间接宾语。在本例中，问题询问的是最后一句中的间接宾语——谁是那个从 Jeff 手里接收牛奶的人。神经网络必须对第五句中所涉及的对象做出精准区分，其中 Bill 是主语，Jeff 是间接宾语，而在第六句中，Jeff 则是主语。当然，我们的神经网络并没有收到任何关于主语或宾语是什么的明确训练，它必须从训练数据的示例中推断出这种合理的理解。

系统必须解决的另一个小问题是理解在整个数据集中使用的各种同义词，如 Jeff 把牛奶“递给（handed）”Bill，但是 Jeff 也可以同样容易地“给（gave）”或“交给（passed）”给 Bill。不过，在这点上神经网络也不必从头开始，因为它可以从词向量那里得到一些帮助。由于神经网络中存在大量的词向量，这些词向量可以给出对应词的定义及其与其他词之间关系的信息，而相似词具有相似的向量化表示，这就意味着神经网络可以将它们视为几乎相同的词。对于词向量化，我们将使用 Stanford 的全局词向量表示（GloVe）。

许多任务都有一个限制，那就是强制上下文包含用于回答问题的确切文字，如在我们的示例中就可以在上下文找到答案“Bill”。我们可以将这一限制当作优势，因为可以在上下文中搜索与最终结果最接近的词。

12.4.2　分析 GloVe 并处理未知令牌

这里我们给出一个 sentence2sequence，它是一个根据 GloVe 定义的映射并将字符串转换为矩阵的函数。该函数将字符串切分为多个令牌（Token，这个过程叫 Tokenization，即切分词或词条），这些比较小的字符串相当于标点符号、单词或单词的一部分。例如，把“Bill traveled to the kitchen.”切分为 6 个令牌，前 5 个令牌对应于单词，最后一个令牌对应于表示句末的“.”。每个令牌都被单独地向量化，从而产生对应于每个句子的向量列表，如图 12-17 所示。

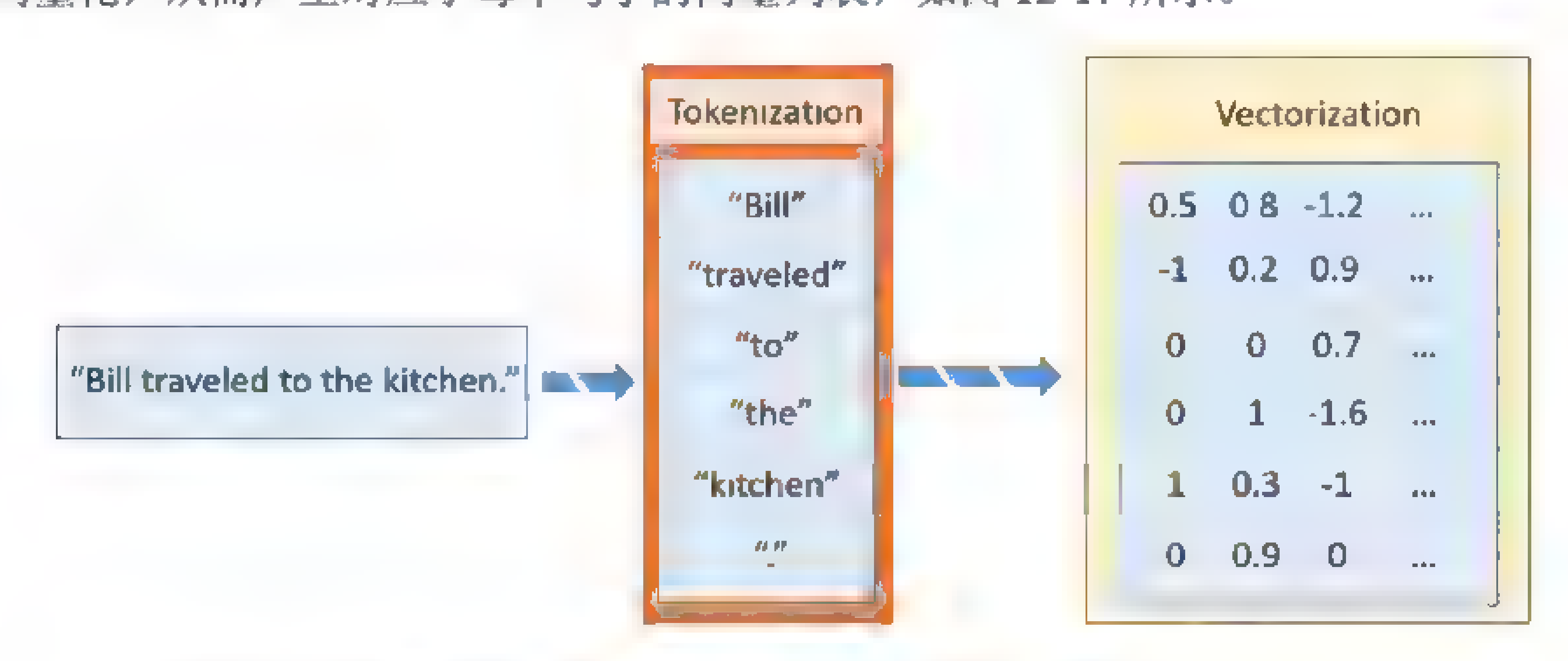


图 12-17　将句子转换为多个向量的过程（图片来源：Steven Hewitt）

在 bAbI 的任务中，系统会遇到一些 GloVe 单词向量化中没有的单词，为了让神经网络能够处理这些未知的词条，我们需要为这些单词维护一个的向量化状态。通常的做法是用单个<UNK>向量替换所有未知令牌，但这不总是有效的。相反，我们可以使用随机化方法来为每个未知令牌创建

一个新的向量。

当第一次遇到一个新的未知令牌时，我们简单地从原始 GloVe 向量化的分布（近似高斯分布）中获取一个新的词向量，并将该向量添加到 GloVe 词映射上。为了收集分布超参数，可以使用 Numpy 中的函数来自动计算方差和平均值。

```
# 反序列化 GloVe 向量
glove wordmap = {}
with open(glove vectors file, "r", encoding="utf8") as glove:
    for line in glove:
        name, vector = tuple(line.split("", 1))
        glove_wordmap[name] = np.fromstring(vector, sep="")

wvecs = []
for item in glove_wordmap.items():
    wvecs.append(item[1])
s = np.vstack(wvecs)

# 收集分布超参数
v = np.var(s,0)
m = np.mean(s,0)
RS = np.random.RandomState()

#而每当我们需要时，fill_unk 会提供一个新的词向量

def fill_unk(unk):
    global glove_wordmap
    glove_wordmap[unk] = RS.multivariate_normal(m,np.diag(v))
    return glove_wordmap[unk]
```

12.4.3 已知或未知的数据部分

其实 bAbI 任务中的词汇量是有限的，这就需要神经网络学好单词之间的关系，即使不知道某些单词的含义，为了加快学习速度，我们也应该尽可能地选择具有内在意义的向量。为此，我们使用贪婪搜索方法来查找存在于斯坦福 GLoVe 词向量数据集中的单词，如果该单词不存在，则用一个未知的、随机创建的词向量表示来替换整个单词。

在这个词向量化模型下，我们可以定义一个新的 sentence2sequence。

```
def sentence2sequence(sentence):
"""
    - 将输入段落转换为(m,d)矩阵，其中 n 是句子中的词条数量，d 是每个词向量的维数.
      这里不需要使用 TensorFlow，因为只是简单地将句子转换为基于映射的序列不需要 TensorFlow
提供计算支持，普通 Python 模块足以完成此任务。
```

```
"""
    tokens = sentence.strip('"(),-').lower().split("")
    rows = []
    words = []
  #对令牌执行贪婪搜索
    for token in tokens:
        i = len(token)
        while len(token) > 0:
            word = token[:i]
            if word in glove_wordmap:
                rows.append(glove_wordmap[word])
                words.append(word)
                token = token[i:]
                i = len(token)
                continue
            else:
                i = i-1
            if i == 0:
      # OOV 单词
                rows.append(fill_unk(token))
                words.append(token)
                break
    return np.array(rows), words
```

现在我们可以将每个问题所需的所有数据打包在一起，包括上下文、问题和答案的向量。在 bAbI 中，上下文被定义成了含有序列号的句子，使用反序列化(contextualize)函数来完成该任务。问题和答案在同一行，用制表符分隔开，所以我们可以利用制表符来标记某一行是否指向了问题。当编号重置时，未来的问题将引用新的上下文（注意，通常一个上下文对应多个问题）。答案还包含我们已保留但不需要使用的另一条信息：参考顺序，与回答问题所需的句子相对应的数字。在我们的系统中，神经网络将自学需要哪些句子来回答问题。

```
def contextualize(set_file):
"""
读入问题数据集并构建问题+答案 - >上下文集。
      输出是一个数据点列表，每个数据点都是一个包含以下内容的 7 元素元组：
          上下文中的句子以向量化形式出现
          上下文中作为字符串词条列表的句子
          向量化形式的问题
          作为字符串词条列表的问题
          向量化形式的答案
          作为字符串词条列表的答案
          用于支持当前未使用的语句的数字列表
```

```
"""
    data = []
    context = []
    with open(set file, "r", encoding="utf8") as train:
        for line in train:
            l, ine = tuple(line.split("", 1))
            # 将行号从所指向的句子中分离出来
            if l is "1":
                # 因为新的上下文总是从 1 开始，所以这是一个重置上下文的信号
                context = []
            if "\t" in ine:
                # 制表符是问题和答案之间的分隔符，在上下文语句中不存在
                question, answer, support = tuple(ine.split("\t"))
                data.append((tuple(zip(*context))+
                             sentence2sequence(question)+
                             sentence2sequence(answer)+
                             ([int(s) for s in support.split()],)))
                # 多个问题可能涉及相同的上下文，所以我们不重置它
            else:
                # 上下文句子
                context.append(sentence2sequence(ine[:-1]))
    return data
train_data = contextualize(train_set_post_file)
test_data = contextualize(test_set_post_file)
```

12.4.4 定义超参数

此时，我们已经充分准备好了训练数据和测试数据，下一个任务是构建用于理解数据的神经网络。我们先从清除 TensorFlow 默认计算图开始，如果想更改一些东西，就可以选择再次运行神经网络。

```
tf.reset_default_graph()
```

由于这是实际神经网络的开始，因此还要定义网络所需的所有常量，称为“超参数”，它们定义了网络的“外观（结构）”和训练方式。

```
# 用于存储在网络中循环层之间传递的数据的维数
recurrent_cell_size = 128

# 词向量中的维数
D = 50

# 神经网络学习的速率。若速率过高的话，则可能会遇到数值不稳定或其他问题
```

```
learning_rate = 0.005
# dropout 概率
input_p, output_p = 0.5, 0.5

# 一次训练问题的个数
batch_size = 128

# 情景记忆的传递次数
passes = 4

# 前馈层大小：用于存储从前馈层传递的数据的维数
ff_hidden_size = 256

# 奖励情景记忆的稀疏性，但会使训练变慢，不要让它大于 learning_rate
weight_decay = 0.00000001

#每次训练时神经网络训练问题的个数，有些问题会被多次计算
training_iterations_count = 400000

#在每次验证之前需要进行训练的迭代次数
display_step = 100
```

12.4.5 神经网络结构部分

神经网络结构被松散地划分为 4 个模块，在 2015 年 Ankit Kumar 等人撰写的文章《*Ask Me Anything: Dynamic Memory Networks for Natural Language Processing*》中给出了相应的说明。

由于神经网络设计了一个循环层，其能够基于文本里的其他信息被动态地定义，因此被称为动态记忆网络（DMN）。DMN 是基于对人类如何回答阅读理解型问题的理解。首先，人类会阅读上下文并在其中建立对相关事实的记忆，记住这些事实后再去阅读问题并重新检查上下文，特别是寻找与问题相关的一些答案的信息，并将问题与每个事实进行比对。

有时一个事实会引导我们走向另一个事实。在 bAbI 数据集中，神经网络希望找到足球的位置，它会先搜索关于足球的句子，并发现 John 是最后一个接触足球的人。然后搜索关于 John 的句子，发现 John 曾经在卧室和走廊里待过。它一旦意识到 John 在走廊上是最后一个人，就可以回答这个问题并自信地说足球在走廊上，如图 12-18 所示。

在每一集或片段中，神经网络都会关注新的事实，以便能够找出正确答案。Kumar 注意到神经网络错误地将一些权重值放在了第 2 句中，这是有原因的，因为 John 已经在那里了，尽管当时他没有足球（Ankit Kumar 等，2015）。

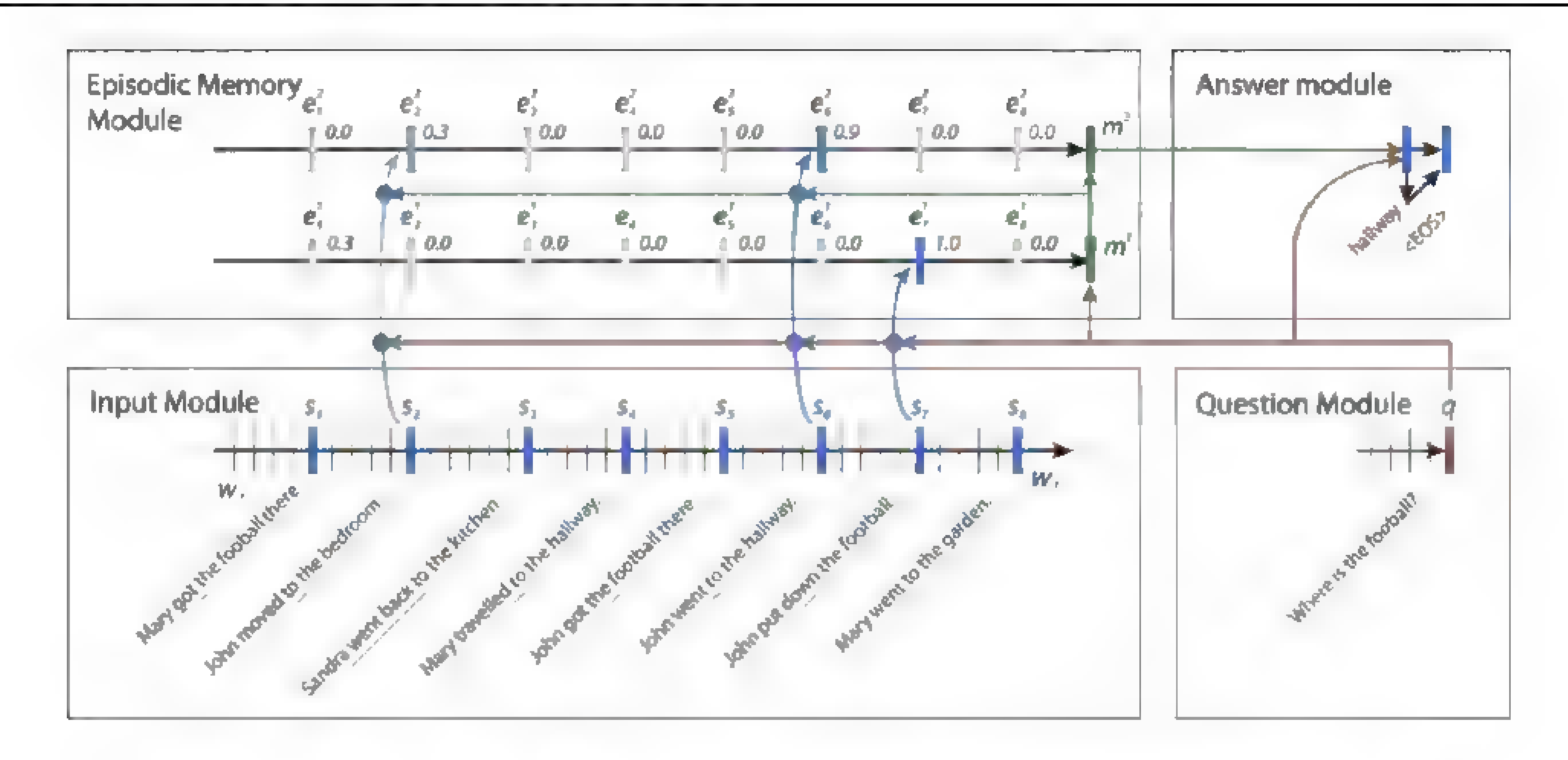


图 12-18　神经网络内部 4 个模块被组合在一起工作以回答 bAbI 数据集中的问题

12.4.6　输入部分

输入模块是动态记忆网络 4 个模块中第一个给出答案的，它包含一个简单的输入，带有一个门控循环单元或 GRU（TensorFlow 的 tf.contrib.nn.GRUCell）的输入管道并以此来收集相关证据。每一个片段的证据或事实，在上下文中都对应一个句子，并且由该时间步长的输出表示，这需要一些非 TensorFlow 的预处理。因此，我们可以收集句子末尾的位置并将其传递给 TensorFlow，以便在后面的模块中使用。

我们可以利用 TensorFlow 中的 gather_nd()函数对这些数据进行处理，并使用这些被处理过的数据来选择相应的输出（gather_nd()函数是一个非常有用的工具）。

```
# cs: 从上下文收集的事实
cs = tf.gather_nd(input_module_outputs, input_sentence_endings)
```

12.4.7　问题部分

问题模块是第二个模块，也可以说是最简单的模块。它包括另一个 GRU 的管道。这次是在处理问题的文本上而不是寻找一些证据，我们可以简单地进到结尾状态，因为数据集保证问题的长度是一个句子。

12.4.8　情景记忆部分

第三个模块是情景记忆模块，就是事情开始变得有意义的地方。它使用注意力执行多个过程，每个过程都包含多个 GRU 并对输入进行迭代。每个过程中的迭代都是基于当时对相应事实的关注程度并对当前记忆的权重进行更新。

12.4.9 注意力部分

神经网络中的注意力最初是为了进行图像分析而设计的,特别是对于图像的某些部分信息远比其他部分的信息与分析主题更具相关性的情况。在执行任务的过程中,神经网络使用注意力能够在深入分析时确定所需对象的最佳位置,如查找图像中目标对象的位置、跟踪在图像之间移动的对象、面部识别或其他需要在图像中寻找最相关信息的任务。

这里的主要问题是注意力或至少硬性注意力(Hard Attention,只关注一个输入位置)很难被优化。与大多数神经网络一样,我们的优化方案是计算损失函数相对于输入和权重的导数,由于其二进制的性质,硬性注意力根本不可微分。因此,我们不得不使用基于实数的变体,被称为软性注意力(Soft Attention),它对所有输入的位置使用某种形式的权重来表明相应的注意程度。而且,这里涉及的权重是完全可微分的,可以被正常训练。虽然学习硬性注意力也是可以的,但它比软性注意力更难以实现且有时表现更差,因此这里我们使用软性注意力。不用担心对导数函数进行编码,TensorFlow 的优化方案已经为我们做了相关处理。

在这个模型中,我们通过构造每个事实、当前记忆和原始问题之间的相似性度量来计算注意力。注意,这里的计算方法与普通注意力的计算方法是不同的,普通注意力只构造事实与当前记忆之间的相似性度量。我们将结果推送进一个两层的前馈神经网络来获得每个事实的注意力常数,然后对输入的事实使用一个 GRU 来赋予权重(通过相应的注意力常数赋予权重),进而修改当前的记忆。为了避免在上下文短于矩阵的全长时向记忆中添加错误的信息,我们创建了一个掩码层,当事实不存在的情况下,模型不必在意它(即保留相同的记忆)。

另一个值得注意的方面是,注意力掩码层几乎总是围绕着神经网络的一个层对表示层进行封包。对于图像而言,该封包最有可能发生在卷积层上(最有可能是直接映射到图像的位置)。而对于自然语言来说,该封包最有可能发生在循环层上。虽然将注意力集中在一个前馈层上,在技术上是可行的,但是通常没有什么用处——至少在那些后续的前馈神经网络层上难以模拟这种方式。

```
def attention(c, mem, existing_facts):
    """
    自定义注意力机制

    c: 张量 [batch_size, maximum_sentence_count, recurrent_cell_size]包含上下文中的所有事实

    mem: 包含当前记忆的张量[batch_size, maximum_sentence_count, recurrent_cell_size]。为了得到准确的结果,对所有事实都应该有相同的记忆

    existing_facts: 张量[batch_size, maximum_sentence_count, 1],就是我们上面提到的(二元)掩码

    """
        with tf.variable_scope("attending") as scope:
            # attending: 度量我们决定去关注什么
```

```
        attending = tf.concat([c, mem, re_q, c * re_q,  c * mem, (c-re_q)**2,
(c-mem)**2], 2)

        # m1: 前馈网络的第一层权重乘法
        # 我们平铺权重值以便手动传递，因为 tf.matmul 从 TensorFlow 1.2 开始就不会自动传
递批量矩阵乘法
        m1 = tf.matmul(attending * existing_facts,
                    tf.tile(w_1,  tf.stack([tf.shape(attending)[0],1,1])))  *
existing_facts
        # bias_1: 第一个前馈层的掩码变体仅对现有事实产生偏差

        bias_1 = b_1 * existing_facts

        # tnhan: 第一非线性。选择 relu 而不是 tanh 等类似的非线性，是为了避免当 tanh 之类
返回值接近 1 或-1 时出现低梯度量级的问题
        tnhan = tf.nn.relu(m1 + bias_1)

        # m2:前馈网络的第二层乘法权重值
        m2 = tf.matmul(tnhan, tf.tile(w_2, tf.stack([tf.shape(attending)[0],
1,1])))

        # bias_2: 第二个前馈层偏差的掩码变体
        bias_2 = b_2 * existing_facts

        # norm_m2: 第二层权重值的标准化版本，用于确保 softmax 非线性不饱和
        norm_m2 = tf.nn.l2_normalize(m2 + bias_2, -1)

        # softmaxable: 在原本密集的张量上使用 sparse_softmax 的技巧。我们让 norm_m2
成为稀疏张量，使其在操作之后再次致密
        softmax_idx = tf.where(tf.not_equal(norm_m2, 0))[:,:-1]
        softmax_gather = tf.gather_nd(norm_m2[...,0], softmax_idx)
        softmax_shape = tf.shape(norm_m2, out_type=tf.int64)[:-1]
        softmaxable    =    tf.SparseTensor(softmax_idx,     softmax_gather,
softmax_shape)
        return
tf.expand_dims(tf.sparse_tensor_to_dense(tf.sparse_softmax(softmaxable)),-1)
```

12.4.10 答案模块

最后一个模块是答案模块，它使用完全连接层对问题和情景记忆模块的输出进行回归并获得“最终结果”的词向量，而与该结果距离最近的上下文中的词便是我们的最终输出（确保结果是一个真实中的词）。我们通过为每个词创建一个“分数”来计算最接近的词，该分数表示最终结果与

当前词的距离。虽然可以设计一个返回多个词的答案模块，但是对于我们要解决的 bAbI 任务来说是没有必要的。

12.4.11 模型优化

梯度下降是神经网络模型的默认优化器，其目标是减少神经网络的“损失”，这是衡量网络性能有多差的一个指标。它通过找到在当前输入下损失相对于每个权重值的导数，然后通过“降低”权重值来减少损失。大多数时候，这种方法效果很好的，但却不是最理想的方法。目前有许多不同的模型优化方法，如使用“动量”或其他更直接路径的近似的方法来最优化权重，而这些优化方法中最有用的一种被称为自适应矩估计（Adaptive Moment Estimation，Adam）。

Adam 方法集成了 AdaGrad 和 RMSProp 算法的优点，通过计算梯度的一阶矩估计和二阶矩估计来为各种不同的参数创建独立的自适应性学习率。进一步讲，Adam 方法计算了梯度的指数移动均值，使用两个超参数（beta1 和 beta2）控制着这些移动均值的衰减速度。而移动均值的初始化值和 beta1、beta2 两参数值接近于 1 时，矩估计的偏差就会接近于 0。若移动均值初始化为零（有些参考资料这里给出的初始化值为 1，笔者认为不准确）和两个超参数（beta1、beta2）值接近于 1 时，就会引起向矩估计的偏差接近于零。

为了抵消这种偏差，Adam 计算偏差校正的矩离估计，其值一般情况下会大于原始值，然后使用校正的估计值来更新整个神经网络的权重值。这些估计的组合使得 Adam 成为整体优化的最佳选择之一，尤其是对于复杂神经网络优化而言。这对于非常稀疏的数据（比如在自然语言处理任务中常见的数据）尤其适用。

在 TensorFlow 中，我们可以通过创建 tf.train.AdamOptimizer 来使用 Adam。

```
optimizer = tf.train.AdamOptimizer(learning_rate)
#我们一旦有了一个优化器，我们就要求它将模型的损失降到最低，以便进行适当的训练
opt_op = optimizer.minimize(total_loss)
```

12.4.12 训练模型并分析预测

一切准备就绪后，我们就可以开始批处理训练数据以训练我们的神经网络。在进行训练过程中，我们应该持续监测神经网络在准确性指标方面做得如何。我们从测试数据里取出一部分作为验证数据集，这样它们和训练数据就不存在重叠问题了。

使用基于测试数据的验证集，可以更好地了解神经网络的泛化能力，即从训练数据里学习到的东西能否应用于其他未知的上下文中。如果在训练数据集中的部分数据上进行验证，那么神经网络可能会出现过拟合现象。换句话说，学习到了特定的示例并记住它们的答案，将无益于神经网络回答新问题。

经过一些训练之后，让我们来看看从神经网络中得到了什么样的答案。在图 12-19 中（笔者这里只给出三个问题的图像，其余两个问题的图像读者可以在代码文件中查看），我们可以将注意力可视化到上下文中所有句子（列）和每个情景（行）上；较深的颜色表示模型对该情景中那个句子

有更多关注。

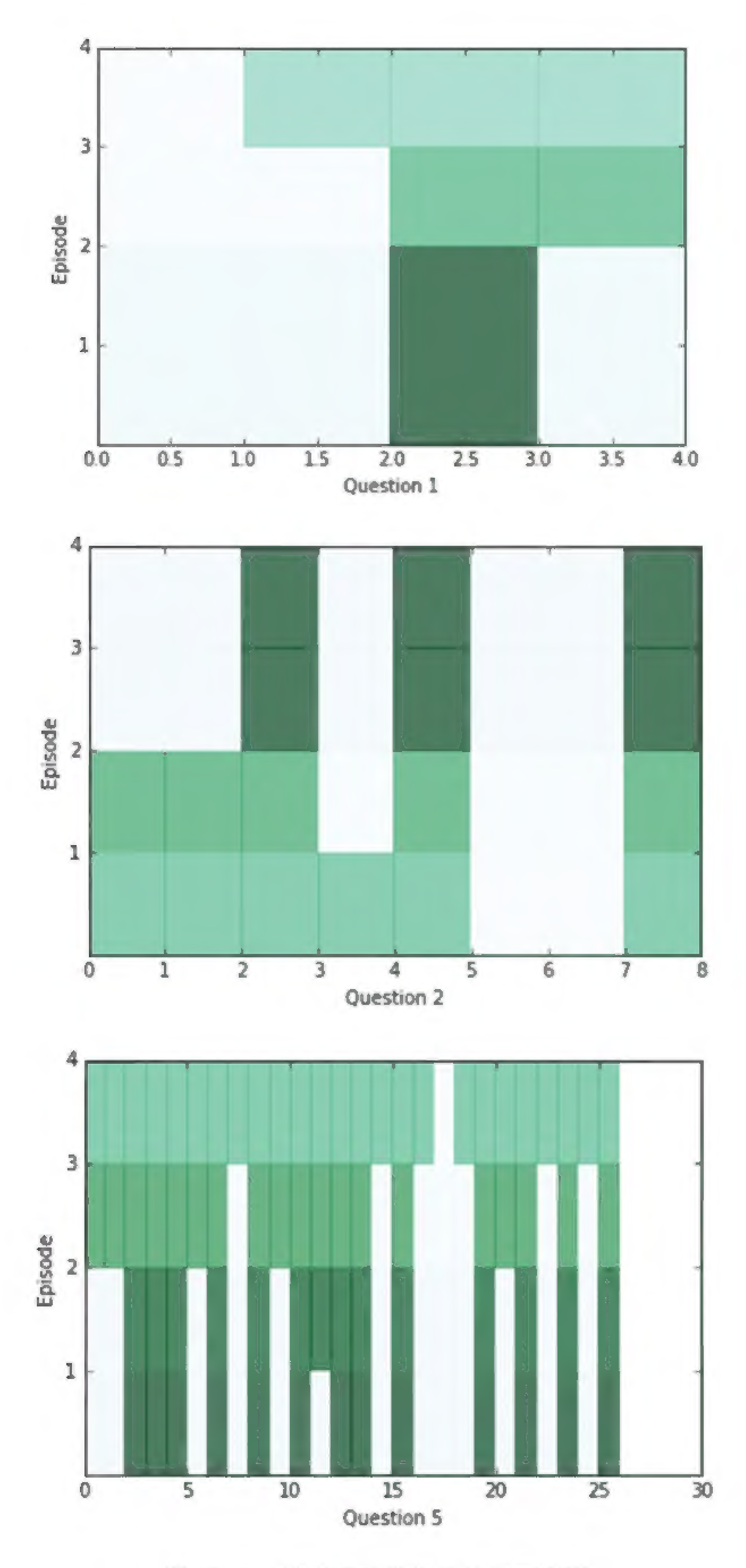


图 12-19 模型输出的部分答案示意图

对于每个问题，应该看到注意力在至少两个情景里发生过变化，但有时注意力可以在一个情景

内找到答案，有时需要全部 4 个情景才有可能找到答案。如果注意力看起来是空白的，那么它可能是饱和的，应该注意到了所有的事项。在这种情况下，可以尝试一个更高的 weight_decay 来训练，以减少甚至阻止这种情况的发生。在训练后期，饱和变得非常普遍。

为了了解上述问题的答案，我们可以使用上下文中距离分数的位置作为索引并查看该索引处的词。

为了得到更好的结果，我们可能需要长时间的训练（一般情况下需要 12 小时左右），最终应该能够达到非常高的精度（超过 90%）。对 Jupyter Notebook 有经验的用户应该知道，在任何时候只要保持相同的 tf.Session，都可以中断训练，并且仍然可以保存网络训练的进展。如果希望可视化注意力和当前网络给出的答案，这种方式很有用。

一旦查看完模型返回的内容，就可以关闭会话以释放系统资源。

12.5　总结

本章简单介绍了基于深度学习的问答方法，特别是对知识库和机器理解的问答。使用深度学习的优点是其可以将所有文本跨度（包括文档、问题和潜在答案）转换为向量（即嵌入处理）。因此，所有文本都可以在统一的语义空间中处理，进而基于符号表示的传统 QA 方法中存在的语义鸿沟问题可以在一定程度上得到缓解，并且这种方法使得 QA 系统可以以端到端的方式构建。所以，现有复杂的基于管道的 QA 过程可以用更直接或更简单的方式代替，预计结果会有所改善。

基于深度学习的 QA 模型也存在许多挑战。例如，现有的神经网络（如 RNN 和 CNN）仍然不能精确地捕获给定问题的语义含义。特别是对于文档，文档中的主题或逻辑结构不能通过神经网络轻松建模，而且在知识库中嵌入项目没有有效的方法。此外，QA 中的推理过程很难通过向量之间的简单数值运算来建模。这些问题是质量保证任务面临的主要挑战，未来应引起更多关注。